University of Alberta

An Adaptive Hybrid Server Architecture for Client-Server Object Database Management Systems

by

Kaladhar Voruganti ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2001

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-60355-5

Canada

University of Alberta

Library Release Form

Name of Author: Kaladhar Voruganti

Title of Thesis: An Adaptive Hybrid Server Architecture for Client-Server Object Database Management Systems

Degree: Doctor of Philosophy

Year this Degree Granted: 2001

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Kaladhar Voruganti

Kaladhar Voruganti
5240 MillCreek Ln
San Jose, California
USA, 95136

Date: Nov 3rd/2000.

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **An Adaptive Hybrid Server Architecture for Client-Server Object Database Management Systems** submitted by Kaladhar Voruganti in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

M. Tamer Özsu
Supervisor

Ronald C. Unrau
Co-Supervisor

Duane A. Szafron

Mario A. Nascimento

Bruce F. Cockburn

Michael J. Franklin
External Examiner

Date: Nov 2nd/2000

To Swami, Nana, Amma, Pavan, Sairaj, Ryan and Sujji

Abstract

The use of object database management systems (ODBMSs) has increased over the past decade due to their ability to model complex data. ODBMSs are used in many important application domains such as electronic commerce systems, medical information systems, telecommunication systems, web document authoring systems, and computer-aided design and manufacturing systems. ODBMSs typically employ the data-shipping client server architecture in which the clients cache data and operate on the cached data. This architecture reduces network latency and increases resource utilization at the client. Currently, there is a lack of consensus amongst the proponents of ODBMSs as to which data shipping architectures and algorithms should be used to implement an ODBMS. For instance, there is a lack of agreement regarding the best data transfer, cache consistency and recovery algorithms. The absence of both robust (with respect to performance) algorithms, and a comprehensive performance study comparing the competing algorithms are the key reasons for the lack of agreement about the desirable client-server architecture.

This dissertation addresses both of these problems. It first presents an adaptive hybrid client-server architecture which utilizes adaptive data transfer, cache consistency and recovery algorithms to improve the robustness (with respect to performance) of a client-server system. The adaptive algorithms presented here can be also used by the existing client-server architectures to improve their performance. Second, this dissertation presents a comprehensive performance study which evaluates the competing client-server architectures and algorithms. The study verifies the robustness of the new adaptive hybrid client-server architecture and provides new insights into the performance of

the different competing algorithms and architectures.

Acknowledgements

I want to thank Dr. Özsu and Ron for all their moral, technical, and financial support. I especially want to thank them for always having faith in me. Without their help, professionally I would not be where I am today. The three of us got along well and we were an excellent team.

I want to thank Paul Iglinski, Srinivas Padmanabhuni, Iqbal, Wade, Rasit, Yuri, Vincent and Anne Nield for all their help and support.

I want to thank Dr. Szafron for always having time to listen to my numerous talks and for helping me better understand OO modeling techniques. I want to thank Dr. Cockburn and Dr. Nascimento for reviewing the thesis and giving me useful feedback. Finally, I want to thank Dr. Michael Franklin for both being the external reviewer, and more importantly, for being a pioneer in the client-server DBMS research area. Your style of research excited me and it made me pursue database systems research.

I want to thank my late father for putting the thought that I should pursue PhD. Words simply cannot explain my gratitude towards my mother. Throughout my life she has always been there for me. I am what I am today because of her. This degree would not have been possible without your her support.

I want to thank my brother Pavan for helping me get closer to GOD. I want to thank Ryan, our Golden Retriever, for giving me unconditional love and always cheering me up. I want to thank our son Sairaj for bringing me good luck. I submitted my first paper after his birth and it was accepted into VLDB.

I want to thank my wife Sundari who sacrificed a lot in her life for my

sake. She always gave me courage and helped me get over my down times. I am lucky to have a wife like her and words cannot explain my gratitude. This degree would not have been possible without her support.

Finally, I want to thank GOD for everything. He has helped me realize that PhD is more than just learning how to do research. Instead its purpose is to build a man's character. I want to pay off my debt to GOD by humbly serving all of mankind.

Contents

1 Introduction 1

1.1 Background and Motivation 3

1.1.1 Client-Server Architecture 5

1.1.2 System Components Under Consideration 5

1.2 Motivation for Adaptive Architectures 8

1.2.1 Adaptive Data Transfer 10

1.2.2 Adaptive Cache Consistency 11

1.2.3 Adaptive Recovery 11

1.3 Dissertation Contributions 12

1.4 Applicability of the new algorithms for emerging architectures 15

1.5 Dissertation Organization 16

2 Background 18

2.1 ODBMS Client-Server Related Work 18

2.2 Related Areas 30

2.2.1 Client-Server File Systems 31

2.2.2 Client-Server Relational Database Management Systems 31

3 Adaptive Hybrid Server Architecture 33

3.1 Motivation for Adaptive Architectures 34

3.1.1 Motivation for Adaptive Data Transfer 34

3.1.2 Motivation for Adaptive Cache Consistency 37

3.1.3 Motivation for Adaptive Recovery 38

3.1.4 Motivation for a Hybrid Server Architecture 39

3.2 Client Buffer Management 40
3.3 Server Buffer Management 42
3.4 Pointer Swizzling 44

4 Data Transfer **46**
4.1 Data Clustering 46
4.2 Intuition Behind Adaptive Data Transfer Mechanism 48
4.2.1 Data Transfer Factors 49
4.2.2 Overview of Adaptive Data Transfer Mechanism 50
4.3 Adaptive Data Transfer Mechanism 52
4.4 Performance Results Overview 65

5 Cache Consistency **67**
5.1 Cache Consistency Overheads 68
5.2 Adaptive Callback Locking (ACBL) 69
5.3 Adaptive Optimistic Concurrency Control (AOCC) 72
5.4 AACC Algorithm 74
5.5 Intuitive Description of AACC 75
5.6 AACC Detailed Description 78
5.7 Deadlock Processing Analysis 81
5.8 Hybrid Granularity Concurrency Control 84
5.8.1 Concurrency Control for Hybrid Servers 85
5.9 Performance Results Overview 87

6 Recovery **89**
6.1 Recovery Background 90
6.1.1 Data Structures 90
6.1.2 Recovery Processing Modes 91
6.1.3 Client-Server Recovery Extensions to ARIES 94
6.2 Hybrid Server Recovery Solution 95
6.3 Updates Performed at both Clients and Server 99

7 Experimental Setup **102**

7.1 System Setup 102

7.1.1 Client Process 106

7.1.2 Server Process 108

7.1.3 Disk Process 110

7.1.4 Network Process 111

7.2 Workload 112

7.2.1 Server Work Allocation 118

7.3 Simulator Validation 118

8 Performance Study **122**

8.1 Cache Consistency Study 123

8.1.1 Cache Consistency Study Outline 123

8.1.2 Private Workload Experiments 126

8.1.3 Shared-HotCold Workload 128

8.1.4 HiCon Workload 146

8.2 Integrated Performance Study 148

8.2.1 Integrated Study Outline 153

8.2.2 Large Client and Large Server Buffers 155

8.2.3 Small Client and Large Server Buffers 158

8.2.4 Large Client and Small Server Buffers 165

8.2.5 Small Client and Small Server Buffers 169

9 Conclusions and Future Work **171**

9.1 Cache Consistency Study Conclusions 172

9.2 Integrated Study Conclusions 174

9.3 Future Work 177

Bibliography **180**

A Glossary **187**

List of Figures

1.1 Client-Server DBMS Architecture 4
1.2 Client-Server System Components 6
1.3 Client-Server System Component Options 9

2.1 A decade of research into page and object server ODBMSs . . 19
2.2 DBMS Cache Consistency Algorithms 22

3.1 ODBMS Client-Server Architecture Classification According to Data Transfer Mechanism 35
3.2 RPT/ROT Data Structures 41

4.1 Different Locality Combinations 47
4.2 Adaptive Data Transfer 52
4.3 RPT/ROT Data Structures 55

5.1 Cache Consistency Scenarios 70
5.2 Deadlock Scenarios 82
5.3 Coupling between Locking and Data Transfer 85
5.4 Logical Lock Segment 86

7.1 Simulator Setup 103
7.2 System Parameters 105
7.3 Workload Parameters 112
7.4 Data Sharing Patterns 113
7.5 Traversals in OO7 113
7.6 Workload Generator Pseudo-Code 119
7.7 Simulator Validation 120

8.1 Private Workload: Low Spatial Locality 127
8.2 Private Workload: Low Spatial Locality 127
8.3 Private Workload: High Spatial Locality 128
8.4 Slow CPU: Sh-HotCold 129
8.5 Message Count 130
8.6 Abort Rate 130
8.7 Fast CPU: Sh-HotCold 132
8.8 Experiments 3 and 4 Cost Breakdown 132
8.9 Zero Abort Variance 134
8.10 Zero Abort Variance Cost Breakdown 135
8.11 Small Server Buffer 135
8.12 Small Server Buffer and 0% Abort Variance 137
8.13 Small Client Log Buffer 137
8.14 Fast Network 139
8.15 Slow Network 140
8.16 Fast Disks 141
8.17 Small Client Buffer 142
8.18 High Spatial Locality 143
8.19 50% Server Work Allocation 144
8.20 Network Delay 145
8.21 HiCon Workload 146
8.22 HiCon Abort Rate 147
8.23 HiCon Costs 147
8.24 ODBMS Classification According to Data Transfer 148
8.25 Systems Under Comparison 149
8.26 Large-Large Buffer Setup 156
8.27 Large-Large Buffer Setup: Private Workload 156
8.28 Large-Large: Sh-HotCold Workload 157
8.29 Small-Large Good Access Locality 159
8.30 Good Access Locality Cost Breakdown 159
8.31 Small-Large Bad Access Locality 160
8.32 Bad Access Locality Cost Breakdown 161

8.33 Small-Large High Temporal Locality 162
8.34 Varying Network Speeds 164
8.35 Large Page Good Access Locality 164
8.36 Large Page Bad Access Locality 165
8.37 Server Buffer with Medium Contention 166
8.38 Installation I/Os 166
8.39 Highly Contended Server Buffer 168
8.40 Installation I/Os 168
8.41 Small-Small 170

Chapter 1

Introduction

Database management systems (DBMSs) have become an integral part of everyday life. Financial, telecommunication, medical, engineering design, manufacturing, and electronic commerce systems all use databases to manage their data. Good performance, in terms of both high throughput and low response time, is a key requirement for most of these application domains. To obtain good performance, it is the responsibility of database users to fine-tune their database setup. However, the fine-tuning of a DBMS is a difficult task due to the complex interaction between the various components of a DBMS. Adaptive systems that can dynamically adapt to changing workloads and system configurations has been identified as a high priority requirement by the users of DBMSs [BBC+98, Ham99, Gra99].

The focus of this dissertation is on designing adaptive architectures and algorithms for client-server object database management systems (ODBMSs). ODBMSs are becoming increasingly popular due to their ability to model complex data which are required by new database applications. ODBMSs are used by applications that are inherently distributed in nature and, hence, there is a need for them to support data distribution. Fine-grained navigation operations where the application program traverses the components of complex data, are prevalent in ODBMSs. Therefore, they employ a client-server architecture where the clients prefetch and cache data locally to optimize the performance of the navigation operations by reducing network latency. These

are referred to as *data shipping* systems. The key premise of this dissertation is that existing ODBMS data shipping architectures and algorithms are not robust with respect to performance across a diverse set of important workloads and system configurations. Thus, there is a need for new client-server ODBMS architectures and algorithms which dynamically adapt as the situation warrants. The need for adaptive ODBMS systems was recognized since the early days of ODBMSs [DFMV90]; however, not much progress has been made in this regard. The lack of adaptive architectures has led to a situation where there are competing ODBMS architectures with their respective limited strengths, but there does not exist a single ODBMS architecture that satisfies the needs of all of the important ODBMS workloads and system configurations.

This dissertation proposes a new adaptive hybrid server architecture which contains a new data transfer algorithm, a new cache consistency algorithm, and a new recovery algorithm. The resulting architecture attains the strengths of the existing systems while avoiding their weaknesses. Thus, the architecture proposed in this dissertation satisfies the decade old requirement of a robust ODBMS architecture. In this dissertation, overall system throughput in commits-per-second is used to measure the performance of an architecture or algorithm. The unifying theme amongst the proposed algorithms is that they dynamically adapt at run-time. The adaptive data transfer algorithm dynamically decides between sending pages or objects between the clients and the server. The adaptive cache consistency algorithm dynamically decides between operating in a pessimistic (asynchronous) or an optimistic (deferred) manner. The adaptive recovery algorithm dynamically decides between redo-at-server and ARIES-CSA recovery approaches. A hybrid server architecture, where the clients and the server can efficiently handle both pages and objects, is a prerequisite for the adaptive algorithms. The data transfer, cache consistency, and recovery innovations proposed in this dissertation can be used not only by the adaptive hybrid server architecture, but also by the existing client-server architectures.

Another key focus of this dissertation is to better understand the interaction between the different sub-components of a client-server ODBMS. These

interactions are complicated, and, to date, there has not been an integrated multi-user study examining them. This dissertation contains a performance study that compares the adaptive hybrid client-server architecture with many of the current client-server architectures. The study verifies that the adaptive hybrid server architecture is indeed more robust, with respect to performance, than the other architectures. It also provides new insights into the interactions between the different client-server sub-systems.

1.1 Background and Motivation

Data-shipping and function-shipping (also called query-shipping) are the two predominant types of client-server architectures. In data-shipping systems, the clients fetch data from the server into their caches and perform some of the database processing locally. The data-shipping architecture helps the composite object navigation operations prevalent in ODBMSs by prefetching data into the client cache, thereby, reducing the response time. In the data-shipping architecture, more DBMS functionality is present at the clients, and this helps to better utilize the hardware resources present at the client workstations. In function-shipping architectures the clients send query requests to the server. The server processes the queries and returns the query results to the clients. Data-shipping architectures are popular because the clients reduce the probability of servers becoming a bottleneck by off-loading some of the work, and this, in turn, is expected to improve their scalability. As well, data-shipping architectures allow the clients to prefetch (if there is locality) useful data into their caches. Prefetching ammortizes network transmission costs. ODBMSs employ data-shipping because they need to provide support for fine-grained traversal (navigational) operations between objects in the database.

This research focuses on data shipping architectures only. Page servers and object servers are the two types of data shipping architectures that are currently used by most current ODBMSs. In the page server architecture, the server sends physical pages to the clients in order to satisfy the client data requests. In the object server architecture, the server sends logical objects to

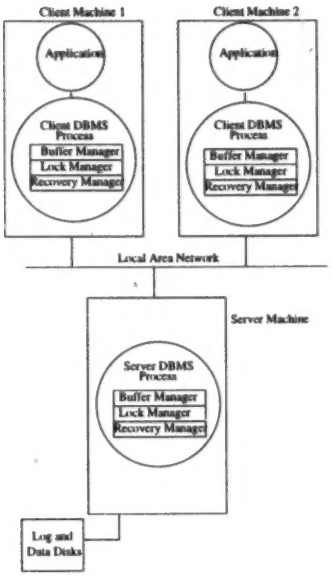


Figure 1.1: Client-Server DBMS Architecture

the clients in order to satisfy the client data requests. Data transfer, client buffer management, server buffer management, cache consistency/concurrency control, pointer swizzling and recovery are some of the important system components that impact the performance of data-shipping client-server systems. Many algorithms have been developed for each of these system components, but these algorithms have been shown to be not robust with respect to different workloads and system configurations [DFMV90, FC94, CFZ94, AGLM95, LAC+96, FCL97, WD94, WD95]. Section 1.2 of this chapter further explains why the existing algorithms and architectures are not robust. In this dissertation, the term *workload* refers to the behavior of the application programs with respect to their data access patterns, and the term *system configuration* refers to the hardware setup (memory, disks, CPU, network) in which the application programs and the ODBMS software operate. This chapter first discusses why these system components are important and then motivates the need for the new adaptive algorithms and the hybrid server architecture.

1.1.1 Client-Server Architecture

In this dissertation, the term *client-server* signifies a data distribution architecture in which a client process and a server process communicate with each other via a network (Figure 1.1). The client process is linked with the application program and the client process requests persistent data from the server process. This work only considers the single server case where multiple client processes interact with a single server on a single machine. That is, it does not consider multiple server or clustered server environments. Client processes do not interact with each other, and they do not store DBMS data or logs on their local disks. The server process is a multi-threaded one that can simultaneously interact with multiple client processes. This dissertation considers both fast (100Mbps) and slow (10Mbps) network interconnections between the clients and the server. Both the server and the client processes contain a buffer manager, a concurrency control manager and a recovery (logging) manager (see Figure 1.2). Even though both the server and the clients understand the notion of objects, the primary focus of this dissertation is on data-shipping architectures and not on architectures where queries are processed at the server. Finally, this dissertation does not consider client-server query processing, query optimization, and indexing issues.

1.1.2 System Components Under Consideration

In data-shipping ODBMSs, clients' data requests are serviced by one or more servers. The server reads data from disk into its buffers and returns them to the requesting client. The clients, in turn, cache data sent by the server in their local buffers and operate on the data. The client subsequently returns updated data back to the server. Figure 1.2 illustrates the different system components that are an integral part of client-server data-shipping architecture. This section briefly describes each of these system components. It is important to note that both clients can request data and locks from the server, but for simplicity, Figure 1.2 only shows client 1's requests. In reality a group of clients could be simultaneously interacting with multiple servers. Server and client data trans-

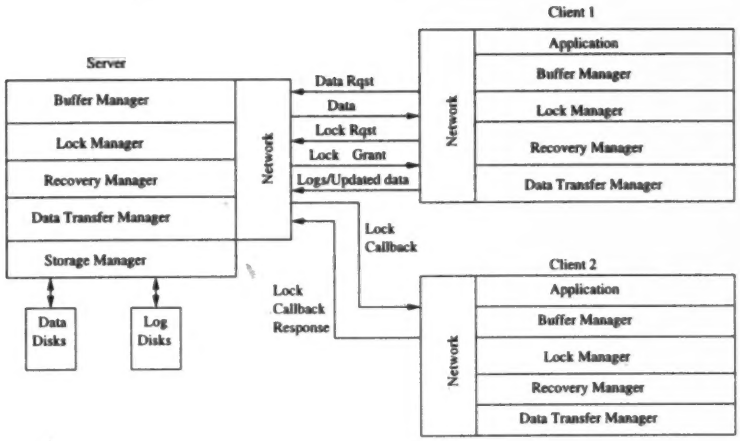


Figure 1.2: Client-Server System Components

fer managers are an integral part of the client-server architecture. The data transfer problem deals with how the server can efficiently transfer data to the clients while satisfying their data requests. It also deals with how the clients can efficiently return updated data back to the server. Previous performance studies have verified the obvious intuition that it is advantageous to maximize the amount of useful data sent in each message and to also minimize the number of messages [DFMV90, LAC+96, CFZ94, OS94a]. Page server and object server architectures employ two different data transfer approaches. Page server architectures try to store well clustered pages on disk and then subsequently send these disk pages to the clients, whereas, (grouped) object server architectures dynamically form object groups, consisting of logical objects, based upon client provided hints (based on expected spatial locality) and send these object groups to the clients. Data clustering refers to how well the user application data access pattern matches the data organization on disk.

Client and server buffer managers also play a critical role. The data that have been retrieved from server disks are cached in both the server and the client buffers. Previous performance studies have shown that the lack of an efficient client buffer manager leads to more client cache misses, and this results

in more client requests and network messages [KK94, Ghe95, FC94]. Lack of an efficient server buffer manager results in more disk I/Os. Buffer organization and the buffer replacement policy are the two key buffer management problems that have to be re-examined within the context of client-server ODBMSs. The data transfer mechanism dictates the structure of the buffers at both the clients and the server.

Since the same data can simultaneously reside in the buffers of multiple clients, it is necessary to keep the client buffers (caches) consistent. That is, within the context of DBMS transaction semantics, each client only operates on the latest committed data. The client caches can be made consistent using a pessimistic locking protocol or optimistic protocols. Therefore, the *concurrency control* and *cache consistency* problems are very tightly coupled. The cache consistency/concurrency control managers at the server and client implement the cache consistency/concurrency control algorithms. Previous performance studies have shown them to have a major impact on the overall client-server system performance [CFZ94, FC94, FCL97, AGLM95]. Pessimistic protocols send explicit lock messages whereas optimistic protocols do not send any explicit lock messages. However, the optimistic protocols encounter more transaction aborts than the pessimistic ones.

In ODBMSs, the applications perform navigation (traversal) operations between the objects by means of object identifiers. The disk version of an object identifier is converted into a memory pointer to allow navigation between objects using memory pointers. This conversion process is known as *pointer swizzling* and is handled by a pointer swizzling manager at the client. The pointer swizzling mechanism is tightly coupled with the data transfer, buffer management, and cache consistency/concurrency control mechanisms. The interaction between the pointer swizzling mechanism and the other data components is discussed in further detail in Chapters 2 and 3. Hardware and software pointer swizzling strategies are two alternatives, and both strategies have their respective strengths and weaknesses [WD94].

Finally, client-server ODBMSs also need to be able to recover from transaction rollbacks and system failures. Thus, it is necessary to assess the impact

of client-server architecture on the existing recovery mechanisms. The client recovery manager generates log records at the clients, which are transferred to the server and stored persistently on server log disks by the server recovery manager. The clients can either return both updated data and log records, only log records (which must then be applied to data pages), or just the updated data pages (which are also logged on log disk). It is important to note that the data transfer mechanism is tightly coupled with the recovery mechanism. For example, if the clients receive objects from the server, then the clients cannot return pages to the server. Thus, the server cannot use a recovery mechanism that requires the presence of updated pages at the server. If the server sends pages to the client, then the client can choose any of the options identified above. The client-to-server data transfer mechanism dictates the type of recovery mechanism that can be employed by the server (like ARIES-CSA or redo-at-server).

In summary, data transfer, buffer management, cache consistency/concurrency control, recovery and pointer swizzling are the key client-server system issues that are examined in this dissertation. These system components are tightly coupled. The remainder of the dissertation analyzes the trade-offs involved in choosing different algorithms for each of the system components. It also proposes new approaches for data transfer, cache consistency and recovery system components.

1.2 Motivation for Adaptive Architectures

Figure 1.3 presents some of the important client-server system components that are considered in this dissertation. Each of the lines in Figure 1.3 connects the popular competing alternatives for a particular system component. Each of the competing alternatives has its strengths and it is intuitively not robust across varying workloads and system configurations. An ODBMS can be constructed by selecting either one end point from each of the system component lines or by an adaptive algorithm that can switch between the competing alternatives (end points). For example, the ObjectStore ODBMS [LLOW91]

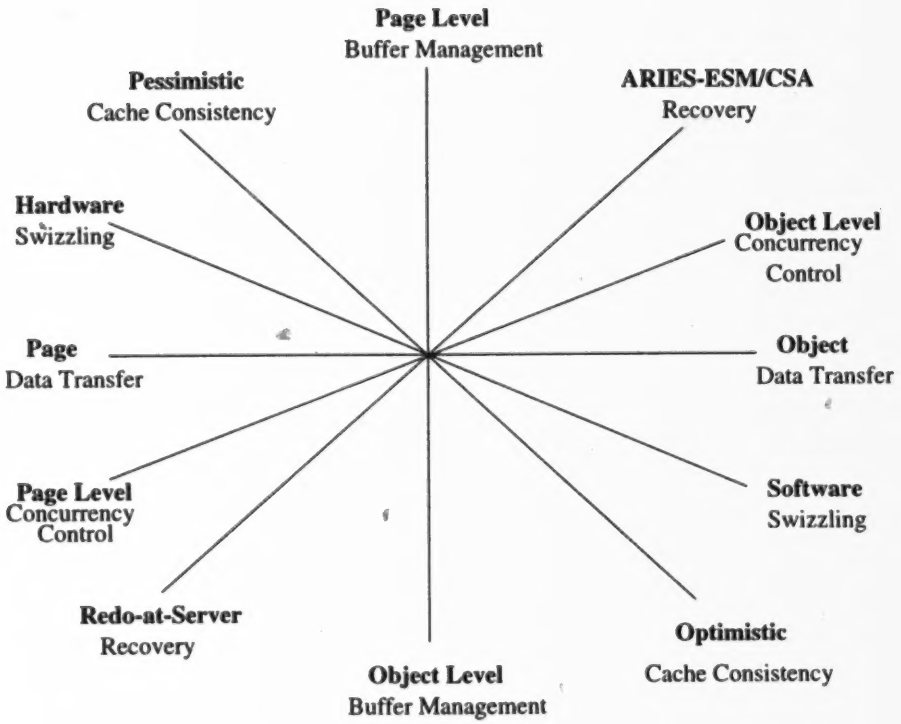


Figure 1.3: Client-Server System Component Options

consists of hardware pointer swizzling, pessimistic cache consistency, page level concurrency control, page level data transfer, and page level buffer management mechanisms. ODBMS application workloads vary with respect to write probability, data sharing patterns between the concurrently executing clients, how well the data access pattern matches the data clustering pattern on disk, and the number of objects accessed within a transaction. The system configuration varies with respect to the network speed, the number of clients, CPU speed, database size, the relative size of the client and server buffers with respect to the the application working sets, and the number of disks.

This section will now briefly motivate the need for adaptive data transfer, cache consistency/concurrency control and recovery algorithms. A more detailed motivation for each of these adaptive algorithms is presented in Chapter 3. Adaptive data transfer, cache consistency/concurrency control and recovery mechanisms are proposed as part of a new hybrid client-server architecture. The hybrid architecture can handle both physical disk pages as well as logical objects. Currently, there already exist buffer management and pointer swizzling proposals that can efficiently handle both pages and objects and these are used by the hybrid server architecture. The details of these buffer management and pointer swizzling mechanisms are also provided in Chapter 3.

1.2.1 Adaptive Data Transfer

From the very beginning of client-server ODBMS research it has been recognized that there is a need for an adaptive data transfer mechanism which can transfer either pages or objects from the server to the client [DFMV90]. It is not efficient to transfer pages from the server to the client when the application data access pattern does not match the way in which data is clustered on disk pages. On the other hand, the efficiency of a grouped object server approach depends upon the accuracy of the object grouping mechanism and it is difficult to design a general purpose object grouping algorithm that is robust under all application data access patterns [DFMV90, LAC+96]. Hence, it would be very useful to design an adaptive data transfer approach which

could dynamically switch between transferring pages or object groups.

1.2.2 Adaptive Cache Consistency

The fundamental problem with current ODBMS client-server cache consistency algorithms is that they can either provide good performance or low abort rate, but not both. Algorithms which provide good performance are optimistic in nature and, therefore, inherently abort prone [AGLM95]. High abort rates are not acceptable in interactive user environments. Similarly, algorithms which provide a low abort rate are too conservative and, therefore, they incur high blocking and messaging overheads [CFZ94]. Thus there is a need for an adaptive cache consistency algorithm that can provide both good performance as well as a low abort rate.

1.2.3 Adaptive Recovery

The two prominent recovery approaches utilized by the client-server ODBMSs are the redo-at-server approach [CDF+94] and the ARIES approach [BP95]. In the redo-at-server approach the logs are sent to the server and are applied by the server to the correct data page. In the client-server ARIES approach, both the logs and the data pages are sent from the client to the server. The problem with the redo-at-server approach is that if the server buffers are contended, it can result in of a high number of reads of the data pages corresponding to the log records which are needed to apply the log records. The problem with the client-server ARIES approach is that it incurs a high network overhead because, even if only a small portion of a page has been updated at the client, the entire page is returned to the server. Therefore, there is a need for an adaptive recovery algorithm which can reduce the problems associated with each of these recovery approaches.

1.3 Dissertation Contributions

This dissertation proposes an adaptive hybrid server architecture that incorporates the following novel features:

- An adaptive data transfer mechanism that dynamically decides whether to ship pages or objects between the server and the client [VÖU99]. The adaptive data transfer mechanism is more robust (with respect to performance) than strictly sending pages or objects. The object grouping component of this data transfer algorithm is more general than the previous object grouping algorithm [LAC+96] in that it can handle multiple page and object sizes.

- An adaptive cache consistency algorithm called *Asynchronous Avoidance-based Cache Consistency* (AACC) [ÖVU98] which provides both good performance and low abort rate. AACC outperforms AOCC and ACBL for important workloads and system configurations. It has been adapted so that it can also be efficiently used by both the hybrid server and the object server architectures.

- An adaptive recovery algorithm [VÖU99] that builds upon ARIES-CSA [MN94]. It can be used not only by the hybrid server architecture that is proposed in this dissertation, but also by object servers and page servers that employ dual page/object buffers at the clients. Finally, the recovery algorithm can also be used by architectures where updates are performed both at the clients and at the server.

It is important to note that the proposed data transfer, cache consistency, and recovery optimizations can also be used by existing object server architectures.

The simulation-based performance study presented in this dissertation compares the hybrid server architecture with the existing page server and grouped object server architectures. The study shows that the hybrid server architecture is more robust than the other client-server architectures. The performance study is important in its own right for the following reasons:

- This is the first multi-user client-server performance study that compares the performance of page servers and grouped object servers. Previous studies either focussed on single-user systems [DFMV90, CDN93, LAC+96], or they did not consider grouped object servers [DFMV90, CFZ94, KK94] or page servers that use a dual page/object buffer at the client [LAC+96] .

- The hybrid server performance study is also the first multi-user client-server performance study that looks at data transfer, buffer management, cache consistency, concurrency control, recovery and pointer-swizzling system components in an integrated manner. These system components are inter-related, and the selection of a particular algorithm for one component has a significant impact on the other components. Currently, the existing ODBMS products use different combinations of algorithms for these system components, and, due to the interaction between them, it is difficult to properly assess the strengths and weakness of the different architectures under a range of important workloads.

- The performance study comparing AACC with ACBL and AOCC reverses the commonly held belief that asynchronous cache consistency algorithms do not outperform synchronous cache consistency algorithms such as CBL [WR91]. Moreover, the previous results indicating that an optimistic high abort algorithm, such as AOCC, is superior to ACBL [AGLM95] might lead one to believe that high abort rates are necessary in order to obtain high performance in client caching systems. However, this study shows that a low abort algorithm such as AACC can outperform AOCC for the most common client caching workload and system configuration. This performance study also helps clarify the performance characteristics of ACBL and AOCC. An earlier study shows that AOCC performs better than ACBL [AGLM95], but that study does not consider workloads where application processing is performed at both the client and the server, or cases where the transaction state does not completely fit in the client cache, or environments when the network experiences

delays (similar to those present in WANs). One would expect the performance of consistency algorithms to be affected in these situations. Therefore, this dissertation evaluates the performance of AACC, ACBL and AOCC for these newer system and workload configurations.

- The new data transfer, cache consistency, pointer swizzling, and recovery contributions presented in this dissertation in conjunction with the integrated performance study have resulted in the following new insights:
  - A page server mechanism with a dynamic dual buffer management mechanism is more robust, with respect to performance for various clustering scenarios, than a grouped object server. Previously it was shown that grouped object servers are more robust than a page server without a dual buffer [LAC+96].
  - The redo-at-server recovery approach in conjunction with the modified object buffer can compete with ARIES type recovery mechanisms. Previously, it was shown that the ARIES style recovery algorithms are more scalable than the redo-at-server type recovery algorithms [WD95].
  - Object servers with the coarse-grained locking mechanism proposed in this dissertation can now efficiently use a pessimistic locking algorithm. Previously, it was thought that pessimistic locking approaches can only be used by page servers [CFZ94].
  - A previous performance study on buffer management has shown that it is better to return updated pages to the server [OS94b]. A subsequent study on server buffer management has shown that it is better to return updated objects to the server [Ghe95]. The performance results presented in this dissertation resolve these conflicting viewpoints.

1.4 Applicability of the new algorithms for emerging architectures

The use of the new data transfer, cache consistency and recovery algorithms is not limited to only the client-server ODBMS domain. Instead, these algorithms with proper modifications can be also utilized by the new emerging client-server architectures. The algorithms developed in this dissertation can be used by the emerging architectures such as the world-wide web, mobile and hybrid function-shipping/data-shipping in the following manner:

- **Data Transfer:** The adaptive data transfer mechanism will be most useful in mobile environments. Since the mobile environments have low network bandwidth and battery power constraints it is important for mobile clients to not transmit and receive poorly clustered pages. Transferring poorly clustered pages increases network latency and client battery power consumption in comparison to transferring isolated objects from poorly clustered pages. Since the adaptive data transfer algorithm tries to send only relevant data across the network it will be useful in mobile environments.

- **Cache Consistency:** The adaptive cache consistency algorithm (AACC) developed in this dissertation will be useful in mobile, web, and hybrid function-shipping data-shipping environments because it has a better combination of low abort rate and high performance than the existing client-server cache consistency algorithms. Re-execution of aborted transactions at mobile clients increases battery power consumption. Thus, the low abort feature of AACC is important for mobile environments. The low abort feature is also important for web applications that require end user interaction during transaction aborts. Finally, low abort rate is also necessary in environments where the server resources are contended. When the server resources (such as CPU and disks) are contended, thenthe re-execution of an aborted transaction also has an impact on the execution performance of all the other transactions run-

ning at the server. Since contended servers can be present in mobile, web and hybrid function-shipping/data-shipping environments, it is desirable to have algorithms with both high performance and low abort rate characteristics.

- **Recovery:** The object server extension to the ARIES recovery algorithm which has been proposed in this dissertation is useful for mobile environments which prefer to transfer objects during poor clustering. This dissertation also proposes another recovery extension that allows for simultaneous update to an object (within the same transaction) at both the clients and the server. This recovery extension is useful in hybrid function-shipping/data-shipping environments where work is performed both at the clients and at the server. Since both web and mobile architectures can, in turn, be based upon a hybrid function-shipping/data-shipping architecture, this recovery extension is also useful for these domains.

1.5 Dissertation Organization

The remainder of the dissertation is organized as follows:

- Chapter 2 describes the related work that has been performed in the client-server data transfer, cache consistency, recovery, pointer swizzling, and buffer management areas. Chapter 2 also briefly highlights how the client-server ODBMS research is related to other client-server research areas such as distributed file systems and distributed relational systems.
- Chapter 3 presents an overview of the hybrid server architecture that is proposed in this dissertation. The details pertaining to the new data transfer, cache consistency, and recovery algorithms are presented separately in the subsequent chapters.
- Chapter 4 presents the new adaptive data transfer algorithm.

- Chapter 5 describes the AACC cache consistency algorithm as well as the AOCC and ACBL algorithms with which it is compared. Chapter 5 also describes how AACC can be extended so that it can be efficiently used by object and hybrid server architectures.

- Chapter 6 presents the new adaptive recovery algorithm. The client-server recovery background information that is necessary for a complete understanding of the proposed algorithm is also included in this chapter.

- Chapter 7 describes the experiment setup. It contains a description of both the system setup and the workloads.

- Chapter 8 presents a performance study that evaluates the new algorithms proposed in this dissertation. It also contains an integrated performance study that compares the performance of the adaptive hybrid server architecture with the leading client-server architectures.

- Chapter 9 presents the key conclusions of this dissertation. Finally, it discusses how the work presented in this dissertation can be extended.

Chapter 2

Background

This chapter describes the research that has been performed over the past decade in client-server data transfer, cache consistency/concurrency control, recovery, pointer swizzling, and buffer management system components. It then discusses how ODBMS client-server architectures are different from the client-server file systems and client-server relational DBMSs.

2.1 ODBMS Client-Server Related Work

Figure 2.1 summarizes the client-server ODBMS research that has been performed over the last decade on data transfer, cache consistency, buffer management, recovery and pointer swizzling. The following is a discussion of these issues:

- **Data Transfer:** The initial client-server performance study [DFMV90] identified page server, object server and file server architectures as three possible client-server architectures. Each of these architectures uses a different data transfer mechanism. In the page server architecture the server returns physical disk pages to the clients. In the object server architecture the server returns logical objects to the clients. In the file server architecture, the system uses a networked file system to transfer pages from the server to the client. This paper concludes that transferring pages (page servers) is desirable when the data access pattern matches

	Cache Consistency				Buffer Management		Recovery		Pointer Swizzling		Data Transfer	
	Page	Object	Page	Object	Page	Object	Page	Object	Page	Object	Page	Object
Year	Low Abort	High Abort	Low Abort	High Abort								
1990	[WN90]							[KGBW90]	[Moss90]	[Moss90]	[DVFM90]	
1991	[CFLS91]	[WR91]					[FZT+92]		[LLOW91]			
1992	[FC92]											
1993									[KK93]			
1994	[CFZ94]		[CFZ94]		[KK94]		[MN94]		[WD94]		[OS94]	[GK94]
1995		[AGLM95]		[AGLM95]	[Ghe95]	[Ghe95]	[WD95]		[EGK95]		[BP95]	
1996	[FCL96]						[PBJR96]			[LAC+96]		[LAC+96]
1997					[CALM97]							
1998	[OVU98]											
1999			[VOU99]					[VOU99]			[VOU99]	[VOU99]
2000												

Figure 2.1: A decade of research into page and object server ODBMSs

the data clustering pattern on disk because this allows the page server to prefetch objects that will be accessed in the future. It also concludes that the object server architecture is insensitive to clustering, and that write operations are very expensive when using a file server. This study prompted the development of clustering and prefetching techniques for both page and object servers [TN92, GK94, LAC+96]. Clustering studies showed that it is difficult to devise general-purpose static data clustering mechanisms that are robust with respect to performance across a wide range of workloads [TN92]. Therefore, researchers have tried to design general purpose dynamic prefetching mechanisms in which hints are provided to the server to allow it to perform intelligent grouping of pages or objects. Predictor-based and code-based prefetching are two types of prefetching algorithms that have been designed [GK94] to help improve the performance of ODBMSs.

In code-based algorithms, the clients examine the application code and try to insert prefetch statements. However, code-based techniques have not been used in DBMSs because it is difficult to determine the sequence of objects (reference chains) that will be accessed at run-time. For example, when traversing path expressions in ODBMSs, the leaf elements in the path expression cannot be prefetched until the intermediate nodes have been identified. The predictor-based prefetching techniques can be classified as strategy-based, training-based, or structure-based [GK94]. In strategy-based techniques the clients employ a specific programmed strategy to generate prefetching hints. For example, the clients can use the current object's identifier as an input into a function and generate the object identifier of the object to be prefetched. Strategy-based techniques are not used by popular ODBMSs [LLOW91, BDP92, Ver98, CDF+94, BP95] because it is difficult to devise universal strategies that can be used by many applications. In structure-based techniques, the structure of the object hierarchy is used in conjunction with a traversal method (breadth-first or depth-first) to identify the objects that can be

prefetched [CK89, KGM91]. Structure-based techniques are not general-purpose either because the objects might not be accessed according to the structured graph that represents the object hierarchy and, thus, their use is also not prevalent amongst the popular ODBMSs. In the training-based prefetching techniques, the execution runs are monitored and statistics are collected during run-time to predict future access patterns [CKV93, PZ91]. Predictor-based ODBMSs are also not used by popular ODBMSs because they require users to perform training runs using benchmarks representing the user applications.

A general-purpose adaptive grouping mechanism has been proposed for object servers that dynamically changes the size of the group based upon the number of objects used in the previously retrieved object groups [LAC+96]. In this algorithm, the client sends the object group size hint along with the data request. The server logically partitions the page on which the object resides into contiguous segments whose size is equal to the client provided group size hint. The server then returns the segment that contains the client requested object. Their performance study [LAC+96] has shown that this dynamic grouping mechanism allows object servers to outperform page servers. However, it does not handle varying object or page sizes, nor does it handle the case when objects are accessed on a page in a non-contiguous manner. A static hybrid data transfer mechanism has been implemented in Ontos [CDN93] in which, for each object type, the application programmers specify whether they want to deal with objects or pages. A partial hybrid server architecture has been proposed in which the server always sends pages to the clients, but the clients can dynamically choose to return either updated pages or updated objects [OS94a]. This flexibility requires revisions in the concurrency control and recovery mechanisms, but these have not been addressed in the partial hybrid server proposal. The partial hybrid server architecture study has suggested that it is best if the clients always returned updated pages to the server. However, this claim is challenged

by the results reported in this dissertation. Another performance study has also shown that, in most cases, it is beneficial to return updated objects from the client to the server [Ghe95]. However, this study did not consider server buffer contention and, therefore, its results are only partially applicable. Furthermore, when dealing with large objects that span multiple pages, it is desirable to be able to transfer only portions of the large object between clients and servers [BP95]. Both page and object servers have to be modified to ensure that the entire large object is not transferred as a single unit. The transfer of data from the client to the server is tightly coupled with the recovery mechanism and will be further discussed in the recovery section below.

	Synchronous	Asynchronous	Deferred
Avoidance Based	CBL [FC94] ACBL [CFZ94]	AACC [This Dissertation]	O2PL [CFLS91]
Detection Based	C2PL [CFLS91]	NWL [WR91]	AOCC [AGLM95]

Figure 2.2: DBMS Cache Consistency Algorithms

- **Cache Consistency:** The DBMS cache consistency algorithms can be classified as avoidance-based or detection-based. Avoidance-based algorithms do not allow for the presence of stale cache data in the client caches, which is permitted in detection-based algorithms. Detection-based algorithms perform commit time validation to check if the transaction has accessed stale objects, and abort if this is the case. Stale data refers to the presence of an older version of data in a client's cache that has been concurrently updated and committed by another client. Avoidance-based and detection-based algorithms can, in turn, be classified as synchronous, asynchronous or deferred, depending upon when they inform the server that a write operation is performed. In synchronous algorithms, the client sends a lock escalation message at the

time it wants to perform a write operation and it blocks until the server grants it permission. In asynchronous algorithms, the client sends a lock escalation message at the time of its write operation but does not block waiting for a server response (it optimistically continues). In deferred algorithms, the client optimistically defers informing the server about its write operation until commit time. In deferred avoidance-based algorithms, the server blocks a client transaction at commit time if the client has updated an object that has been read by other clients [FCL97]. Figure 2.2 depicts this classification along with some of the popular cache consistency algorithms.

In client caching (or data shipping) systems, inter-transaction (across transaction commit boundaries) caching of data and locks is generally accepted as a performance enhancing optimization [FC94, WN90]. A previous performance study has shown that for most workloads, it is preferable to cache read locks instead of both read and write locks across transaction boundaries [FC94]. That is, write locks are downgraded to read locks at the end of a transaction. Upon being informed about the write operation, the server, in turn, tries to either invalidate or update remote client caches by sending them messages. For most user workloads, invalidation of remote cache copies during updates is preferred over propagation of updated values to the remote client sites [FC94]. Furthermore, the ability to switch between page and object level locks is generally considered to be better than strictly dealing with page level locks [CFZ94]. Within the family of avoidance-based algorithms, it has been shown [FC94] that the synchronous callback locking (CBL) algorithm, despite its higher messaging overhead, has similar performance to the optimistic two-phase locking (O2PL) [CFLS91] class of algorithms while incurring a much lower abort rate [FC94]. In O2PL, the write lock escalation message is deferred until commit time, whereas in CBL, the clients send synchronous lock escalation messages at the time of the update operation and do not proceed until they receive a response from

the server. Therefore, CBL encounters lower deadlock abort rate than O2PL as the data contention rate increases.

There are many performance studies comparing avoidance-based and detection-based algorithms [FC94, AGLM95, WR91]. The general conclusions are that synchronous avoidance-based algorithms, such as CBL, are superior to synchronous detection-based (e.g. C2PL) and asynchronous detection-based (e.g. NWL) algorithms. It has been shown that deferred detection-based algorithms (e.g. AOCC) can outperform synchronous avoidance-based algorithms (e.g. ACBL) even while encountering a high abort rate. Avoidance-based cache consistency algorithms encounter deadlock aborts but not stale cache aborts, whereas optimistic detection based algorithms encounter stale cache aborts but not deadlock aborts.

There has also been an attempt at developing a hybrid temperature-based algorithm [CLH97], where the data contention temperature is maintained for each object. If the temperature is high then the clients operate on the object in a pessimistic manner; if the temperature is low, the clients operate on that object in an optimistic manner. However, due to the reactive nature of this algorithm, changing user data access patterns, and dynamic addition and deletion of clients, can potentially lead to high abort rates and low performance. The performance of this approach [CLH97] with respect to AOCC and ACBL is not known.

Most of the cache consistency research has been conducted within the context of page servers. Since it is inefficient to send individual lock escalation messages to lock each object [CFZ94], the proponents of object servers adopted optimistic cache consistency algorithms [AGLM95] where the lock escalation messages are deferred until commit time and are sent along with the commit message. However, optimistic cache consistency algorithms incur higher abort rates and, in many cases, are undesirable from a usability standpoint [AGLM95]. Therefore, currently, there not does exist a cache consistency algorithm for object servers that

provides both high performance and low abort rate.

- **Pointer Swizzling:** Object identifiers are an integral part of object DBMSs. An object identifier uniquely identifies an object in the database; they are system assigned and immutable. They can be either logical or physical. A physical object identifier (POID) stores the physical disk address of the object within the identifier itself, whereas a logical object identifier (LOID) has a level of indirection to point to the object. An intermediate mapping data structure (hash table or B tree) is usually employed to deduce the location of an object from its LOID. LOIDs provide more flexibility with respect to object migration, replication and deletion than POIDs, but they incur mapping overhead that is not present with POIDs.

  The task of converting an object identifier stored on disk into a memory pointer is known as *pointer swizzling.* ODBMSs employ pointer swizzling to improve the navigation operation response time. Pointer swizzling algorithms can be classified in three different (orthogonal) ways as: eager or lazy, hardware or software, and direct or indirect [Whi94].

  In eager swizzling, all the object identifiers present in an object or page are swizzled as soon as the data are loaded into the client cache, whereas in lazy swizzling the OIDs are swizzled only when the objects are actually accessed. Eager swizzling eliminates the need to check whether or not an OID has been swizzled during each OID access. This helps performance since it prevents a check-per-pointer access to see whether the pointer is swizzled, but it can lead to the swizzling of pointers that are never accessed by the application (wasted work).

  In direct swizzling, the source object points directly to the target object via the memory pointer. In indirect swizzling, the source object points to the target object via a level of indirection such as an object table. The level of indirection adds extra overhead during traversal operations, but it also provides the flexibility to efficiently migrate, delete, and change the size of the objects.

In the hardware pointer swizzling approach, the page level virtual memory facilities (page faulting mechanism) provided by the operating system are used to detect when a page has been accessed, and the page is sent from the server and loaded into the client cache. All the pointers present in a page are eagerly swizzled to point to the target virtual memory frames corresponding to the target pages. However, these pages are only brought into the client's cache when the pointers pointing to the page are actually accessed. In the software swizzling approach, a function call interface is provided to the client applications to access the pointers. The function code performs residency checks and dereferencing of pointers.

QuickStore [WD94] and ObjectStore [LLOW91] systems use the hardware/direct/eager swizzling approach. BeSS [BP95] uses the hardware/indirect/eager swizzling approach. Versant [Ver98], THOR [LAC+96] and O2 [BDP92] use the software/indirect/lazy swizzling approach.

Systems using the hardware/eager swizzling approach store the in-memory version of the object identifiers on disk [WD94, LLOW91]. These approaches need to also store an additional meta-object corresponding to each data page on the disk. This meta-object contains information about the disk address (corresponding to the target object's page) corresponding to each pointer. When the server sends the data page to the client, it also sends the corresponding meta-object to the client. The meta-object is used by the client to determine the disk address of the pages that need to be faulted into the client cache. If the size of the client's working set is larger than the size of available virtual memory at the client, then multiple database pages can map to the same virtual memory frame. At the time when the client is faulting in a page into its cache, it checks to see whether the pointers in that page point to the appropriate target pages. If the client detects such problems, then it changes the pointers to point to new, non-conflicting virtual memory locations. Thus, in addition to the meta-object, the hardware swizzling schemes also maintain a bitmap

object corresponding to each data page persistently at the server. When there is a virtual memory frame conflict, the clients request the corresponding bitmap object from the server to find and reset the pointers in the page. The details of the hardware swizzling approach that store memory pointers on disk have been only briefly mentioned here and can be found elsewhere [WD94].

The hardware swizzling approach, which stores memory pointers on disk, and encounters the inflexibility problems associated with POIDs. The software swizzling approach has the flexibility to use both LOIDs or POIDs, but most object DBMSs use LOIDs because they insulate the applications from object migration and deletion. The advantage of storing memory pointers on disk is that it alleviates swizzling and unswizzling operations. However, storing memory pointers makes it difficult to provide support when clients are executing on heterogeneous operating systems with different pointer sizes and virtual memory management mapping mechanisms. Storing memory pointers on disk in combination with direct pointer swizzling also restricts the size of the database that can be accessed by the client (without encountering integrity problems associated with accessing objects on pages that have been ejected from the client buffer) to the size of the client's virtual memory. Currently, object servers and page servers that want to manipulate data at the object level do not store memory pointers on disk because they do not want to manipulate data strictly at page level using the operating system provided page handling mechanism. Since the hardware/eager swizzling approach relies on operating system provided page faulting mechanism, it usually employs only page level locking, data transfer, recovery and buffer management mechanisms.

- **Buffer Management:** In a client-server system, the server buffer is used to store data that has been retrieved from disk and sent to the clients. The client buffer is used to cache useful data across transaction boundaries to reduce the number of data requests to the server.

Both the server and the clients contain log buffers in addition to data buffers. The data buffers are usually managed using the least-recently-used (LRU) page replacement policy, and the log buffers are managed using the first-in/first-out (FIFO) replacement policy when the log buffer is full. The logs in the log buffer are also flushed if the corresponding data item is flushed from the data buffer, or if the transaction has initiated a commit operation. Over the past decade, buffer management innovations have been made for both client and server buffers. Dual buffer management techniques can be utilized by clients in page server architectures to increase client buffer utilization [CALM87, KK94]. Dual buffering allows caching of both well clustered pages and isolated objects from badly clustered pages. Dual buffers can be partitioned either statically [KK94] or dynamically [CALM87]. Both object and page servers can use the modified object buffer (MOB) at the server to store the updated objects returned by the clients [Ghe95]. The MOB helps in the batching of updates sent by the client. That is, if the clients send many updated objects from one page to the server, the server can perform a single read of the page corresponding to the updated objects from disk. The MOB also allows the server to intelligently schedule installation reads (reads that are explicitly performed for installing an updated object on its home disk page) using a low priority process which amortizes the installation read cost [Ghe95]. In client-server architectures, if the client caches are large with respect to the client working set, and there is not much data sharing between the different clients, then the server buffer acts more like a staging buffer [FC94]. In such situations, it is better to use a buffer replacement policy such as *LRU with hate hints* [FCL92] instead of the standard LRU for managing the server buffers. In LRU with hate hints the server marks those pages that are present in client caches as *hated*. The pages with the *hate* marks are ejected first from the server buffer in order to increase the overall buffer utilization of the entire client-server system by reducing the number of duplicates (data present in both server and client caches).

- **Recovery:** In client-server DBMSs, log records are generated at the clients during the execution of an application program. There are trade-offs as to when these log records should be generated, how they should be generated, where they should be stored persistently (locally on client disks or at the server), and how they should be transfered to the server (if not stored persistently on local client disks) [WD94, PBJR96, MN94, FZT+92].

  If the server does not persistently store the client generated log records, then the server has to rely on the clients during its restart recovery. Even though storing client generated logs only on local client disks reduces the work that has to be performed by the server [PBJR96], this solution is unacceptable in most client-server environments because it is not desirable for reliable servers to rely on potentially unreliable clients to recover from server failures.

  If the server is managing log disks, then the clients can return only the updated page (whole-page logging) [WD95], return both updated pages and log records (ARIES approach) [MN94, FZT+92], or return only the log records or updated objects (redo at server) [WD95]. Therefore, in the redo-at-server approach, the clients can return either updated objects, or log records. A previous performance study [WD95] has shown that the whole-page logging approach saturates the log disk, because it is inefficient to log the entire page when only a small portion of the page has typically been updated. The study also showed that the redo-at-server approach suffers from the installation read problem as the number of clients increases. Thus, the study advocated returning both pages and logs to the server.

  It has also been shown that for ODBMS workloads, it is not desirable to generate a log record for each update since the same object can be updated multiple times within a transaction. Instead, it is more efficient to perform a *difference* operation at commit time between the before-update and after-update copies of data and to generate a single log record

[WD95].

Existing page server recovery mechanisms use the Steal/No-Force buffer management policy as supposed to No-Steal/No-Force, Steal/Force, and No-Steal/Force policies [ÖV99]. In the Steal/No-Force buffer management policy, the pages in stable storage (disks) can be overwritten before a transaction commits, and pages do not need to be forced to disk in order to commit a transaction. Steal/No-Force is generally regarded as the most efficient buffer management policy [MN94], but the published object server recovery proposals [KGBW90] do not use it. The need for an efficient object server recovery algorithm has been identified as an outstanding research problem [FZT+92, MN94].

Some of the existing page server client-server recovery algorithms do not allow for the simultaneous update of a page by multiple clients [MN94, FZT+92], which is allowed by others [PBJR96]. In client-server architectures, the clients have the option of not playing a role [FZT+92] or actively participating [MN94] during transaction rollback [FZT+92]. Moreover, both servers and clients can also initiate a checkpoint operation [MN94]. Client checkpoints can be more frequent than the server checkpoints, and thus, help in reducing the amount of log that needs to be examined during client failures. The three-pass ARIES recovery system that was developed for centralized DBMSs can also be used to recover from distributed client and server failures [MN94].

2.2 Related Areas

The client-server data distribution paradigm has been extensively studied within the context of file systems and relational database management systems (RDBMSs), and the benefits of distributed systems, such as autonomy, reliability, performance and scalability, are well known. Therefore, it is important to gain an understanding of the similarities and the differences between client-server ODBMSs and these related fields where different assumptions

have been made about key factors such as workload characteristics, correctness criteria and data manipulation granularity. A detailed study on client-server ODBMS caching provides a useful comparison between client-server ODBMSs, client-server file systems and client-server relational DBMSs [FCL97].

2.2.1 Client-Server File Systems

Client-server file systems generally operate in environments where the user access patterns are mostly sequential and concurrent access conflicts are rare [Fra93]. This, in turn, leads to the design of simple sequential page prefetching and coarse-grained concurrency control algorithms. File systems do not provide the atomicity, consistency, isolation and durability (ACID) criteria which are required by database applications. Most of the file systems do not provide support for read-write conflicts and they allow for situations where crashes can result in lost updates [Fra93]. The traditional notion of database transaction management is absent in these systems and this is usually left as the responsibility of the application program. File systems deal with data transfers, concurrency control, and data consistency at the level of pages (usually a group of pages representing a file).

File systems do not satisfy the needs of the database applications because these applications can operate in both high contention as well as low contention environments. It is very important for database applications to have an underlying storage system which enforces the ACID (Atomicity, Consistency, Isolation, Durability) properties of transactions. Finally, the manipulation of data strictly at the file level or page level (with respect to security and locking) is inappropriate for many database applications.

2.2.2 Client-Server Relational Database Management Systems

Relational database management systems (RDBMSs) have been designed to provide support for both sequential as well as non-sequential workload sce-

narios. RDBMS architectures provide support for both coarse-grained (table/page level) as well as fine-grained (record level) manipulation of data. Thus, RDBMSs satisfy the needs of DBMS applications more closely than distributed client-server file systems. However, RDBMS applications mostly perform set-oriented queries comprising of select/project/join operations. This usually involves the selection of results after the processing of a large amount of data. The trend in RDBMSs has been to employ the *function-shipping* (also known as *query-shipping*) client-server architecture [Fra93]. For set-oriented queries, the function-shipping architecture reduces the communication overhead because the server is able to process the query and return only the query results back to the clients. This architecture is also desirable if the clients have a limited amount of hardware resources and if the clients and the servers are connected via networks having high latencies. However, in function-shipping architectures, the increased load at the server can potentially cause scalability problems. Since most of the processing in the function-shipping architecture takes place at the server, the client processes are light weight. Moreover, this architecture can employ centralized DBMS buffer management, recovery, and concurrency control algorithms.

The function-shipping approach is not adequate for fine-grained navigational workloads that are present in ODBMSs because prefetching and client caching techniques are usually not adopted in this approach. This generates too many individual function (query) requests between the client and the server (if the queries are not batched) and causes an increase in the traversal response time due to the network latency. Thus, the function-shipping architecture by itself is not adequate for client-server ODBMSs. The current trend is for ODBMSs to provide both navigational and query processing functionality. Researchers are in the initial stages of exploring hybrid function-shipping/data-shipping architectures [KJF96, HKU99]. This dissertation does not deal with hybrid function-shipping/data-shipping architectures, but the algorithms developed in this dissertation can be potentially used by these architectures.

Chapter 3

Adaptive Hybrid Server Architecture

This chapter introduces the adaptive and hybrid client-server architecture proposed in this dissertation. This new architecture consists of the following components:

- **Adaptive Data Transfer:** The server and the clients dynamically decide whether to transfer pages or objects among themselves.
- **Adaptive Recovery:** The system dynamically decides to operate in either ARIES mode or in redo-at-server mode. Furthermore, the recovery mechanism is hybrid because it can handle the case when there are either pages or objects present in the client cache.
- **Adaptive Cache Consistency:** The clients and the server dynamically decide whether to send synchronous, asynchronous or deferred lock escalation messages.
- **Hybrid Buffer Management:** As the clients and the server can transfer both pages and objects among themselves, the client and server buffer management components must be able to handle both pages and objects.
- **Hybrid Concurrency Control:** As the clients and the server can manipulate data at either the page or the object level, it is necessary to be

able to lock data at both these levels. Moreover, the concurrency control mechanism can dynamically escalate and de-escalate between page and object-level locks. Concurrency control algorithms which can dynamically switch between page and object-level locking have been previously developed for page servers [CFZ94].

- **Software Pointer Swizzling:** Since the client cache might contain either pages or objects, the pointer swizzling mechanism cannot manage data solely at the page level. Since the hardware pointer swizzling mechanism relies on operating system provided page level support, it cannot efficiently provide object-level buffering or concurrency control. Therefore, the software pointer swizzling mechanism is used by the new architecture. The software pointer swizzling mechanism that is used by the SHORE [CDF+94] ODBMS is used by the hybrid server architecture.

This chapter first provides the motivation behind adaptive data transfer, adaptive cache consistency and adaptive recovery mechanisms. The details pertaining to these three adaptive components are presented separately in subsequent chapters. This chapter then describes the client and server buffer management, as well as the pointer swizzling mechanisms used by the hybrid client-server architecture proposed in this dissertation.

3.1 Motivation for Adaptive Architectures

3.1.1 Motivation for Adaptive Data Transfer

In current client-server systems, the servers ship either physical disk pages or logical objects to the clients. Systems where the servers ship physical disk pages to the clients are known as *page servers* (O2 [Sof98], ObjectStore [LLOW91], BeSS [BP95], SHORE [CDF+94]): systems where the servers ship logical objects to the clients are known as *object servers* (THOR [LAC+96] and Versant [Ver98]). Page server systems allow clients to return either updated pages or updated objects to the server whereas, object servers restrict clients to

return only updated objects. Figure 3.1 classifies some of the current ODBMSs

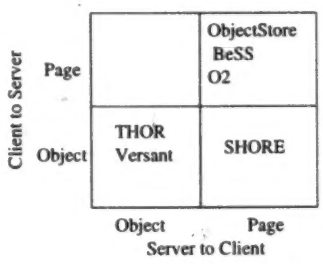


Figure 3.1: ODBMS Client-Server Architecture Classification According to Data Transfer Mechanism

according to their data transfer mechanism. Since the data transfer granularity from the server to the clients is the primary distinguishing factor between page servers and object servers, the adaptive data transfer discussion in this section is based on the data transfer mechanism of object and page servers. Page server systems can outperform object servers when the application data access pattern matches the data clustering pattern on disk (which is referred to in the rest of the dissertation as *good clustering*) [DFMV90]. By receiving pages under good clustering, the clients in the page server architecture are able to exploit spatial locality, and, thus, prefetch objects that they will likely use in the future. Spatial access locality helps page servers to amortize communication costs. In comparison, object servers incur higher communication overhead since they transfer individual objects from the server to the client [DFMV90]. However, when the data clustering pattern on disk does not match the data access pattern (bad clustering), transferring the entire page from the server to the clients is counter-productive because this increases the network overhead and decreases client buffer utilization since only a few objects on the page are referenced. Dual client buffer approaches in which the client buffer is partitioned into an object buffer segment for managing objects and a page buffer segment for managing pages have been proposed to improve client buffer utilization. Dual client buffer mechanisms [KK94, CALM87] allow the storage of well clustered pages and isolated objects from badly clustered pages.

Page servers are inefficient for the emerging hybrid function-shipping/data-

shipping architectures [KJF96] where, in addition to requesting data from the server, the clients also send queries to be processed at the server. The server processes the queries and returns only the results back to the client. If a query result is spread across multiple disk pages each of which contains only a few objects, then it is inefficient to send all of the disk pages to the client [DFB+96].

An important study on clustering [TN92] has shown that it is difficult to come up with good clustering when multiple applications with different data access patterns access the same data. Therefore, good clustering cannot be taken for granted, and the problem of transferring badly clustered pages is a fundamental issue in page servers.

The data transfer problem of object servers that transfer single objects can be partially resolved by transferring a group of objects rather than a single object from the server to the client. A dynamic object grouping mechanism has been proposed [LAC+96] that makes grouped object servers competitive with page servers with respect to the data transfer mechanism. However, this technique considers single fixed page size (28K page) and small object sizes (50 to 100 byte objects). Therefore, there is a need for a more general object grouping mechanism which can handle varying page and object sizes.

As evident from the above discussion, there is definitely a need for an adaptive server-to-client data transfer mechanisms because page servers perform well during good clustering and object servers perform well during bad clustering. Similarly, there is also a need for an adaptive client-to-server data transfer mechanism which can dynamically switch between returning to the server updated pages or updated objects. If only a few objects on a page have been updated, then systems that return updated pages to the server incur higher network overhead in comparison to systems that return updated objects. However, in systems that return updated objects, the server has to re-install these updated objects on their corresponding disk page in the server buffer (redo-at-server recovery) before writing the page back to disk. Hence, if the server buffer is heavily contended, then the home pages might not be present in the server buffers, necessitating reads to retrieve the pages from

disk (known as *installation reads*) [Ghe95].

The proponents of object servers have introduced the notion of a modified object buffer (MOB) [Ghe95] at the server. The MOB stores updated objects that have been returned by the clients. It intelligently (trying to reduce the seek time) schedules a group of installation reads in the background and thus reduces the installation read overhead. A performance study [Ghe95] has shown that the use of a MOB improves the performance of architectures that return updated objects to the server over those that return pages. Another performance study [OS94a] has shown that, when clients update large portions of a page and the server buffers are contended, it is desirable to return updated pages to the server. As evident from the above discussion, there is also a need for an adaptive client-to-server data transfer mechanism.

3.1.2 Motivation for Adaptive Cache Consistency

Client cache consistency is an important problem in distributed ODBMSs. The problem exhibits itself in multi-user systems where data are accessed by and reside in the caches of multiple clients that are connected to the servers via networks. Cache consistency algorithms can be classified as *avoidance-based* or *detection-based* [FCL97]. Avoidance-based algorithms prevent access to stale cache data within a transaction, whereas detection-based algorithms allow stale cache data access, but detect and resolve them at commit time. *Stale data* refers to data in cache that are out-dated due to concurrent committed updates by another client. Adaptive Callback Locking (ACBL) is commonly accepted as the leading avoidance-based cache consistency algorithm [FC94] and Adaptive Optimistic Concurrency Control (AOCC) [AGLM95] is the leading detection-based cache consistency algorithm. These algorithms are discussed in more detail in Chapter 5.

AOCC generally outperforms ACBL, with respect to overall system throughput, in environments where the client cache is sufficiently large to hold the entire transaction state (data and logs) and the application processing is done strictly at the clients [AGLM95]. AOCC achieves this even while encounter-

ing a higher abort rate than ACBL, mainly due to its efficient abort handling mechanism.

One might conclude that AOCC is a superior cache consistency algorithm since its performance is generally better than ACBL. However, performance is not the only issue: the high abort rate of AOCC makes it unsuitable for interactive application domains. Furthermore, it is necessary to evaluate how a high abort rate affects AOCC performance in environments where the application processing is performed not only at the clients but also at the servers (hybrid architectures) and when the entire transaction state cannot fit into the client cache. Hybrid architectures, where queries are sometimes executed at the client by caching the necessary data and sometimes executed at the server by shipping queries to the server, are emerging as the desirable client-server DBMS architectures [KJF96]. Transaction state cannot fit into the client cache when large transactions access many objects, or transactions access large objects (e.g. multimedia), or when multiple user processes share the client's cache.

These observations suggest that there is a need for algorithms which provide good performance while maintaining a low abort rate. Although an optimistic algorithm such as AOCC can outperform ACBL, most commercial client caching DBMSs continue to use ACBL (or its variants) because they also have to support applications which cannot tolerate a high abort rate. Ideally, it is desirable to use a cache consistency algorithm whose performance approaches that of the best (avoidance-based or detection-based) cache consistency algorithm while incurring a low abort rate.

3.1.3 Motivation for Adaptive Recovery

The existing client-server recovery work has been conducted strictly within the context of page server data-shipping systems in which the server sends pages to the clients and the clients return updated pages (and log records) to the server [FZT+92, MN94]. The existing page server recovery work is inadequate with respect to the following important scenarios:

- As motivated in the adaptive data transfer section, there is a need for an architecture in which the server and the clients can transfer both pages and objects among themselves. The existing page server recovery algorithms are inadequate since they only allow for the transfer of pages between the server and the clients. Thus, there is a need for an adaptive recovery mechanism which makes it possible to have an adaptive data transfer mechanism.

- If object updates are performed at both the clients and the server, then the existing client-server recovery mechanisms are inadequate because they do not handle the case where the same object has been successively updated both at the client and at the server within the same transaction [MN94]. Thus, the current recovery mechanisms need to be enhanced for this situation.

The details of the recovery terminology and page server recovery mechanism are discussed in detail in Chapter 6.

3.1.4 Motivation for a Hybrid Server Architecture

This dissertation proposes a new hybrid server architecture. The hybrid server architecture is a prerequisite to the adaptive data transfer mechanism proposed in this dissertation. Since the adaptive data transfer mechanism can transfer both pages and objects between the clients and the servers, the hybrid server architecture needs to efficiently handle both pages and objects. The data transfer mechanism dictates the types of algorithms that can be used by client buffer management, server buffer management, pointer swizzling, concurrency control and recovery system components. Therefore, the adaptive data transfer mechanism makes it necessary for these different system components to be hybrid in nature.

Since both the clients and the servers can deal with both pages and objects, it is necessary to have dual (page/object) buffers at both the clients and the servers. Currently, there are some ODBMSs which use dual buffers at the

clients (for example, [CDF+94, KK94]), and others that use dual buffers at the server (for example, [LAC+96]). Therefore, the hybrid server architecture proposed in this dissertation uses these techniques. None of the current ODBMSs use dual buffers at both the server and the clients.

Similarly, it is necessary for the hybrid server architecture to efficiently perform cache consistency/concurrency control operations at both page and object level because it deals with both pages and objects. Hence, in addition to providing efficient support for adaptive data transfer, adaptive cache consistency, and adaptive recovery mechanisms, the hybrid server architecture has to also efficiently implement object level concurrency control.

The adaptive data transfer, cache consistency and recovery algorithm details are provided in Chapters 4, 5 and 6, respectively. The client buffer management, server buffer management, and pointer swizzling management mechanisms that are used by the adaptive hybrid server architecture are described in the subsequent sections of this chapter.

3.2 Client Buffer Management

Since the hybrid server architecture deals with both pages and objects, it is necessary for the client buffers to manage both pages and individual objects. Numerous client dual buffer management schemes have been developed in the past (e.g. [KK94, CALM87, OS94a]). Dual buffers allow clients to store both well clustered pages and isolated objects from badly clustered pages. The proposed hybrid server uses a modified version of a known dual buffering scheme [KK94]. The buffer space is partitioned into page and object buffer components. The application program can access objects from both the object and page buffers, each of which is managed using the second-chance (LRU-like) buffer replacement policy [BP95]. To minimize data copying overhead when copying objects from the page buffer to the object buffer, the second-chance buffer replacement algorithm for the page buffer component is enhanced such that preference is given to retaining pages that are well-clustered (more than 60 percent of the page has been accessed [KK94]) by flushing the badly clustered

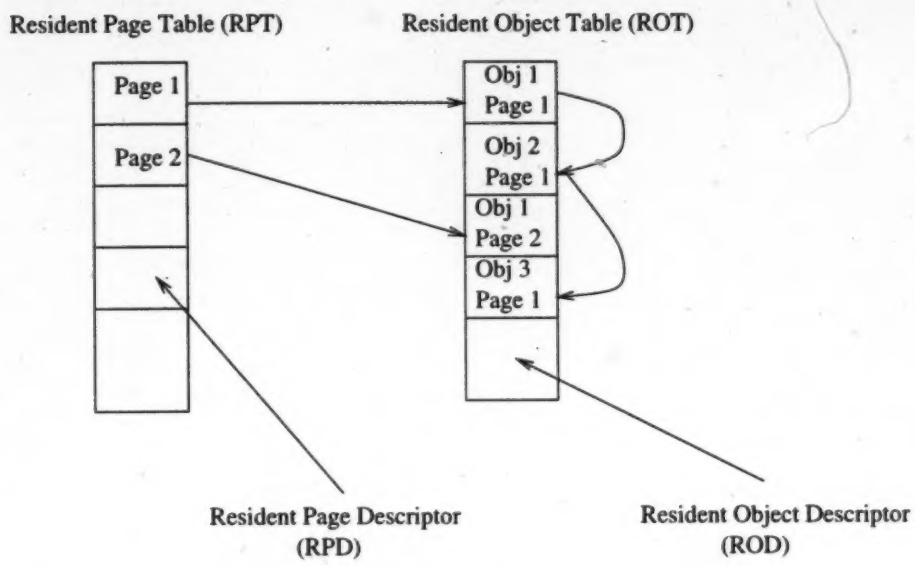


Figure 3.2: RPT/ROT Data Structures

pages before flushing the well clustered pages. To eliminate the number of duplicates present in the page and the object buffer, referenced objects from the badly clustered pages in the page buffer are copied into the object buffer not when the object is initially referenced, but in a lazy manner just before the page is ejected from the page buffer. Moreover, objects are eagerly re-located from the object buffer into their corresponding pages in the page buffer when the page is re-loaded into the page buffer after it is received from the server. The previous dual buffer performance study [KK94] has shown that lazy-copying and eager re-location is the most desirable object handling strategy for dual buffers because it eliminates duplicates in the client cache and, thus, increases client buffer utilization.

To manage the data present in the dual buffers, the client maintains a *resident page table* (RPT) and a *resident object table* (ROT) (Figure 3.2). Each RPT entry is known as a *resident page descriptor* (RPD) and it corresponds to a page, and each ROT entry is known as a *resident object descriptor* (ROD) and it corresponds to an object. The RODs of the objects that belong to a

page are linked together (resident object entry chain) as shown in Figure 3.2.

The dual buffer mechanism that is used in this dissertation partitions the client buffer into page and object buffer segments according to the workload. If the application data access pattern matches how data reside on disk (good data clustering), then it is desirable to allocate a larger portion of the buffer to the page buffer because this minimizes page ejections and copying of the data from the page buffer into the object buffer. Similarly, it is desirable to increase the object buffer size when the clustering is poor because this retains a larger portion of the database and, thus, increases client buffer utilization.

The clients also contain an undo log buffer. Before an object is updated for the first time, a pre-updated copy of the object is copied into the undo log buffer. The pre-updated copy of the object is used to generate log records. The log buffer is managed using a first-in/first-out buffer replacement policy.

3.3 Server Buffer Management

In the hybrid server architecture, the server contains a page buffer and a dual modified hybrid buffer (MHB), and the server buffer has been partitioned equally into these two types of buffer. The server page buffer is used to store the pages that are retrieved from disks, and pages that have to be sent to the clients. The dual modified hybrid buffer stores the updated objects/pages returned from the clients. The MHB is a variant of the modified object buffer (MOB) because the MHB can store both updated objects as well as updated pages, whereas, a MOB only stores updated objects. The server also contains a log staging buffer which stores the logs to be written to the persistent store. The server page buffer is managed using the second-chance (LRU-like) buffer replacement policy [BP95]. To obtain higher server buffer hit rates for workloads with data sharing between the clients, the server buffer manager did not employ hate-hint buffer replacement policies. The MHB and the log buffer are managed using the first-in/first-out buffer replacement policy. The data from the server log staging buffers is flushed either during commit time, or when a log buffer page is full.

When loading an updated page into the MHB, the server checks to see whether the page already exists in the page buffer and in the MHB. If the page already exists in the page buffer, then the server invalidates it and returns its page buffer frame to the page buffer free list. If the page already exists in the MHB, then the server merges the updates of the newer and the older version of the pages in the MHB. When loading the updated objects into the MHB, the server does not eagerly install these updated objects if their page resides in the page buffer. The objects get installed into their corresponding pages only if the MHB is getting flushed, or if the pages corresponding to the updated objects are requested by another client. In the latter case, updated objects are only installed if they have been committed by their corresponding transaction. That is, uncommitted updates are not installed on pages that are sent to other clients. The lazy installation of updated objects from the MHB to their corresponding pages increases the MHB absorption capability[OS94b, Gru97, Ghe95]. MHB absorption refers to the reduction in the number of writes of updated pages to disk due to the grouping of the updated objects (that arrived at the server separately) and writing them to disk via a single write operation.

The data from the MHB buffer is flushed using a background process. When 80 percent of the MHB buffer is full, it triggers the flushing of 10 percent of the MHB buffer. During the flushing process, the server intelligently schedules the disk I/O operations to minimize the I/O (seek and rotational delay) costs. These MHB buffer flushing parameters were empirically determined in a previous study on server buffers [Ghe95]. When flushing a modified page, the server simply flushes the page to disk. When flushing updated objects, the server ensures that all of the objects corresponding to a page that are present in MHB are flushed via a single write operation. The server first schedules an installation read operation, and follows this immediately by the installation of the updated objects on the page, and the writing of the updated page back to disk [Ghe95]. This sequence of read and write operations are carried out for all of the updated objects in the 10 percent of the MHB that is being flushed.

3.4 Pointer Swizzling

The hybrid server architecture uses the software pointer swizzling approach because it efficiently handles both pages and objects. The software pointer swizzling mechanism uses logical object identifiers (LOIDs), and it contains a level of indirection between the source and the target object to allow for object migration and deletion. The LOID to physical object address mapping information is maintained at the client in a hash table. The pointers present in the objects are swizzled when they are accessed for the first time. Only those pointers present in updated data (page or object) being returned to the server are unswizzled. The software pointer swizzling mechanism which is used by the hybrid server is similar to the approach used by the SHORE object storage management system [CDF+94]. In this approach, when the source object tries to access a target object via the target object LOID, the following processing takes place at the client and the server:

- **Processing at the client during initial pointer access:** The client hashes the target object LOID and checks the hash table to see if there is a link to the resident object descriptor (ROD) of the target object from the hash table entry. If the target's ROD exists, then the LOID of the target object present in the source object is swizzled into a pointer that points to the ROD of the target object. The ROD, in turn, contains a pointer to the target object. If the target object's ROD does not exist at the client, then the client sends the LOID to the server.

- **Processing at the Server:** The server satisfies the client request by determining the physical address of the object, from the client supplied LOID, using a LOID-to-disk address mapping data structure. Then the server retrieves the appropriate data page from the disk or the server buffer and returns the page to the client. The server also sends the LOID-to-disk address mappings for all of the other objects residing on the same page or object group being sent to the client.

- **Processing at the client upon receiving the data:** Upon receiving the data, the client creates the RODs and resident page descriptors (RPDs) for the received data and stores the data in the client cache. The client also swizzles the LOID of the target object present in the source object as described above.

- **Unswizzling processing at the client:** The clients perform the unswizzling operation when updated pages or objects are returned to the server. The unswizzling operation is performed before the client generates the log records. The client checks to see whether a pointer has been swizzled and it then obtains the LOID value from the corresponding ROD, and replaces the pointer with the LOID.

Chapter 4

Data Transfer

Page servers [DFMV90] and object servers [LAC+96] are currently the two most prominent client-server ODBMS architectures. Page servers send disk pages from the server to the client, and object servers send logical object groups from the server to the clients. Previous performance studies have shown that the performance of the data transfer mechanisms of both page server and object server architectures suffers for certain important workloads and system configurations [DFMV90, CFZ94, LAC+96]. This chapter presents an adaptive data transfer mechanism which builds upon the strengths of both page and object data transfer approaches while avoiding their weaknesses. The adaptive data transfer mechanism is described in three sections. The first section describes the concept of data clustering, which plays a major role in determining the performance of the different data transfer algorithms. The second section provides an intuitive overview of the adaptive data transfer algorithm. Finally, the third section provides the algorithm details.

4.1 Data Clustering

Data clustering [DFMV90, TN92] refers to how well the application data access pattern matches data placement on disk. Since page servers transfer disk pages from the server to the client, the data clustering pattern is an important performance determining factor for page servers. Transferring well-clustered

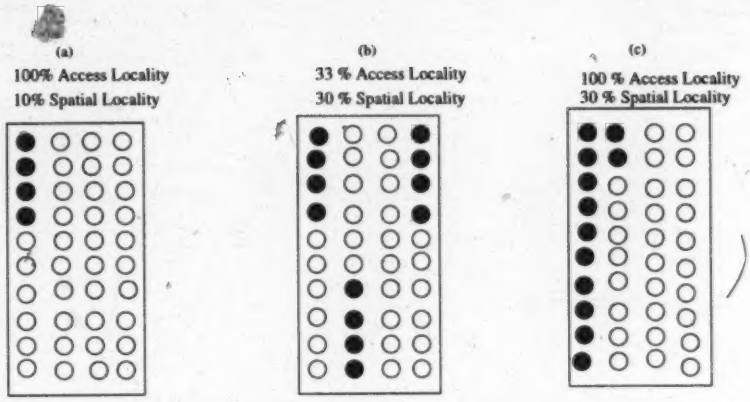


Figure 4.1: Different Locality Combinations

disk pages, which match the application data access pattern (good clustering), helps the page server to take advantage of spatial locality and, thus, prefetch useful objects that will be accessed in the future. However, badly clustered disk pages, which do not match the application data access pattern, degrade page server performance by reducing effective network utilization and client buffer utilization. In this dissertation, data clustering is defined by *spatial locality*, *temporal locality* and *access locality* parameters. These probabilistic values are specified with respect to a particular application. That is, a particular page can be viewed to have both good spatial locality with respect to one application, and bad spatial locality with respect to another application. The definitions of the three locality values are as follows:

- **Spatial Locality:** It is defined as the ratio of the number of bytes accessed in a page to the size of the page. As shown in Figure 4.1(a), spatial locality of 10 percent for a 4 Kilobyte sized page with 100 byte sized objects means 4 objects on the page have been accessed, and, as shown in Figures 4.1(b) and 4.1(c), a spatial locality of 30 percent means 12 objects on the page have been accessed.

- **Access Locality:** It is defined as the ratio of number of contiguously accessed bytes in a page to the total size of the page. For example, for a

4 Kilobyte sized page with 100 byte sized objects, access locality of 100 percent in conjunction with the spatial locality of 30 percent means that 12 contiguous objects on the page are accessed [refer to Figure 4.1(c)] and, an access locality of 33 percent with a spatial locality of 30 percent means that only 4 out of the 12 objects are accessed in a contiguous manner on the page [refer to Figure 4.1(b)].

- **Temporal Locality:** is defined as the probability that the previously accessed bytes on a page will be accessed again to the same page. For example, for a 4 Kilobyte sized page with 100 byte sized objects, a temporal locality of 100 percent in conjunction with 10 percent spatial locality and 50 percent access locality means that both the first and the subsequent accesses to the page (separated by accesses to other pages) access the same 4 objects, whereas a temporal locality of 50 percent means that there is a 50 percent chance that the subsequent accesses to the page will access new objects on the page.

Spatial, temporal and access localities together determine the relationship between the data access pattern and the data placement on disk. These localities together represent how data is clustered and they are specified as percentage values between 0 and 100 percent. Data clustering is an important parameter which plays a key role in determining the performance characteristics of page and object servers. The three locality values described in this section are varied in Chapter 8 to analyze the performance of page and object servers under different clustering scenarios.

4.2 Intuition Behind Adaptive Data Transfer Mechanism

The fundamental principle of the adaptive data transfer mechanism is that it can dynamically adapt between sending pages or a group of objects both from the server to the client, and from the clients to the server. The objective of the

adaptive data transfer mechanism is to correctly switch between sending pages or objects among the server and the clients at run-time, as the workload or system configuration changes, because sometimes it is better to transfer pages and at other times it is better to transfer objects.

Both the client and the server have an important role to play in making the adaptive behavior possible. The adaptive data transfer mechanism is unique because both the server and the clients pass hints among themselves to make this behavior possible. The hints are both simple to compute and are calculated dynamically at run-time. There are four key factors which determine the performance of a data transfer mechanism. These factors are briefly described in the next subsection.

4.2.1 Data Transfer Factors

The four important factors that affect data transfer mechanism performance are:

- **Server-to-client data transfer mechanism:** This factor is important because it determines the network and CPU overhead of sending and receiving data from the server to the client. These overheads impact the time a client has to wait before its requested data arrives from the server.

- **Client-to-server data transfer mechanism:** This factor is important because it determines the network and CPU overhead of returning updated data from the client to the server. These overheads have an impact on the time it takes a client to commit a transaction.

- **Client buffer utilization:** The server to client data transfer mechanism determines the objects that are present in the client cache and this, in turn, determines the client buffer utilization. This factor is important because it determines the number of client cache misses. Client cache misses are expensive because a client cache miss leads to an explicit data request by the client to the server. Furthermore, if the client requested

data does not reside in the server buffer then the server has to perform a disk I/O.

- **Server buffer contention:** This factor is important because it determines the cost of performing an I/O at the server. If the server read buffer is contended then there is a higher probability that a client cache miss will also result in server cache miss. If the server modified object buffer is busy then it increases the probability that installation read/write I/Os will interfere with normal I/Os that are performed to retrieve client requested data. Installation reads are necessary when the client returns updated objects to the server, and the pages corresponding to these objects do not reside in the server buffer, and thus, have to be retrieved from disk. The writing of the updated pages back to disk is known as installation writes. Similar to installation reads, installation writes can also interfere with normal client read request I/Os.

4.2.2 Overview of Adaptive Data Transfer Mechanism

This section now provides a brief overview of the adaptive data transfer mechanism, and in the process it describes how each of the four above mentioned factors are handled. The details of the algorithm are quantified below in the algorithm description section.

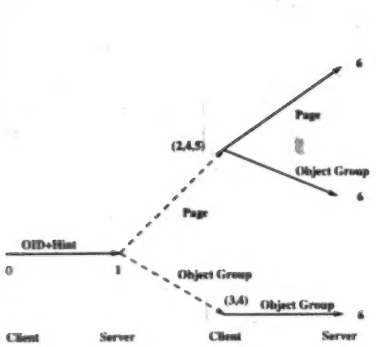
The client initially (during a cold start) sends an object identifier and requests the corresponding data page, on which the object resides, from the server. The server returns the requested page to the client. Upon receiving the page, the client stores the page in its page buffer and keeps track of the number of objects that have been accessed in that page. If the number of used objects is low, and if there is a need to eject the data page from the client buffer due to data contention, the client copies the objects that are in use into its object buffer and ejects the page. The goal of the dual buffering strategy is to try to increase the client buffer utilization and reduce the client cache miss overhead. The client also requests a page from the server when the accessed objects are spread across the page in a non-contiguous manner (low

access locality). In these situations, the adaptive server architecture sends pages from the server to the client with the goal of reducing the number of client cache misses, because the object grouping mechanism at the server forms object groups that consist of contiguously placed objects.

If the client determines that not many of the objects are accessed (low spatial locality), then the client switches and requests a group of objects to try to reduce the server-to-client network overhead by not sending badly clustered pages from the server to the client. Depending upon the application data access pattern, the client dynamically changes the size of the requested object group. The client sends the object group size as a hint along with the data request to the server. The object group size is specified as a percentage of the page size instead of as an absolute number in order to be able to handle variably-sized objects and pages. If the size of the object group starts to increase, then the client switches over to requesting pages in order to try to lower the group forming overhead at the server, and the group disassembling overhead at the client.

When the server receives a client's hint for an object group, the server has the option to override the client hint and send pages if the server determines that its buffers are contended (i.e., if the modified object buffer (MOB) is not able to batch many updates to the pages and is, therefore, performing a high number of installation reads). If the server disk utilization is high, then the installation reads may interfere with normal read operations that are performed to read client requested data. Therefore, the server explicitly informs the clients when it is busy by piggybacking this hint along with other messages. If the page has not already been flushed from the client page cache, then the client uses the server provided busy hint to return updated pages rather than updated objects to the server. However, if the client dual buffer mechanism needs to discard a badly clustered page and retain only useful objects, the client ignores the server provided busy hint. The server and the clients send hints to each other, but have the freedom to override these hints depending upon their local run-time conditions.

If the server buffer is not busy, the client returns updated objects to the



0) Client sends object request and a hint
1) If Server Buffer is Contended
  Send Page (override Client Hint)
 Else
  Use Client Hint to Send Page or Object Group

2) If Page Well Clustered
  Keep Page Intact
 Else
  Extract and Keep Objects

3) Load Objects into Client Buffer

4) Determine whether Next Request is Page or Object Group
 If Object Group then Adjust Group Size
5) If Server MOB absorption is low or a Large Portion of page is updated
  Return Page(s)
 Else
  Return Object(s)

6) Write into Modified Dual Buffer

Figure 4.2: Adaptive Data Transfer

server because it wants to reduce both the network overhead and increase the server MOB absorption. Thus, returning updated objects from the client to the server helps to reduce the client to server data transfer overhead and installation write overhead.

4.3 Adaptive Data Transfer Mechanism

The details of the adaptive data transfer mechanism are as follows (Figure 4.2 gives an overview of this algorithm and each of the points in the figure are described in detail below):

1. **Initial Client Request:** A client's first request is for an object; it sends an object id to the server and requests the corresponding object. The client locally keeps track of the current object group size. It initializes the object group size value to be equal to the page size and sends the hint for a page request along with the object id to the server.

2. **Request Processing at Server:** The server receives the client data request and the client object group size hint. In servicing the request, the server takes the following actions:

- **Server-to-Client Data Transfer Rule:** If the MOB absorption rate is less than 30 percent, the read buffer miss rate is greater than 30 percent, and disk utilization is greater than 80 percent, then the server informs the client that it is busy, and the server ignores the client hint and sends pages to the client.

- **Rule Analysis and Alternatives:** The server expects the clients to use the hint (about the contended server buffer) to return updated pages. If the server receives updates objects from the clients under these conditions then it has to perform installation read operations, and the installation read operations, in turn, interfere with normal read I/O operations (that are performed to satisfy client data requests). Installation read refers to the I/O operation that is performed because the page corresponding to an updated object is not present in the server buffer. The MOB absorption rate refers to the rate at which the MOB is able to batch the installation of updated objects to their corresponding home page. The server read buffer miss rate refers to the number of server read buffer misses. If the clustering is bad, it is the responsibility of the client dual buffer mechanism to discard badly clustered pages and retain only useful objects. Thus, the client can override the server hint if it is desirable for the client to discard the page from its buffer. Installation reads are performed to read the page corresponding to an updated object that is present in the MOB.

  The MOB absorption rate threshold of 30%, read buffer miss rate threshold of 30% and disk utilization rate threshold of 80% were empirically determined.

  A MOB absorption rate that is lower than 30% reduces the ability of the MOB to batch the installation of updated objects on their corresponding home page. Higher MOB absorption rates increases the batching of object update installations on their corresponding pages. This, in turn, helps to reduce the number of installation

read operations. On the other hand, during higher MOB absorption rates it is desirable to receive updated objects because this helps to reduce the number of updated page writes (installation writes) to disk.

It is also necessary to examine the server read buffer miss rate because if the read buffer miss rate is lower than 30%, it reduces the probability that installation reads will interfere with normal client read requests and receiving updated objects from the clients is not a liability.

Finally it is also necessary to check disk utilization in conjunction with the read buffer miss rate and MOB absorption rate because if the disk utilization is lower than 80% then the installation read operations can be performed in the background and they will not be a problem.

- **Threshold Calculations:** The server calculates the *read buffer miss ratio* by keeping track of the number of read buffer misses over the total number of accesses to the read buffer due to client read requests. For values between 25 and 35 percent there was not appreciable difference in the overall system throughput. If a *read buffer miss ratio* threshold that is larger than 30% is chosen then this allows clients to continue sending updated objects to the server; if the MOB absorption is low and the disk utilization is high, then the installation reads will interfere with the normal client read requests. If a *read buffer miss ratio* threshold that is smaller than 30% is chosen, then this aggressively favors sending pages from the server to the client and expects the clients to return updated pages even though the server is not performing many data request I/Os and therefore, there is a lower probability for the installation reads to interfere with normal read I/Os. The server calculates the MOB absorption rate by checking to see whether there are other objects belonging to the same page as the client returned updated objects

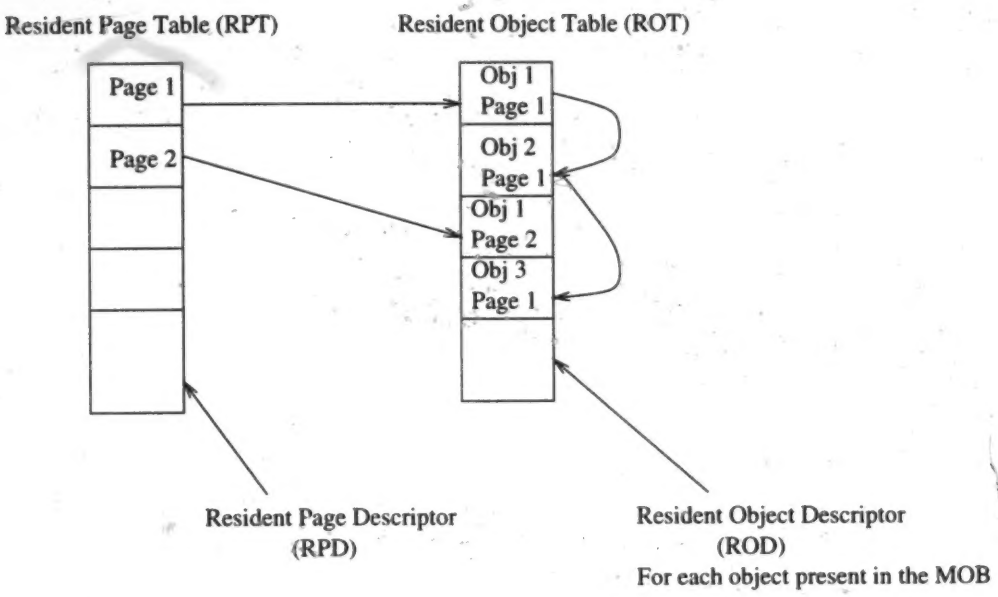


Figure 4.3: RPT/ROT Data Structures

already present in the MOB. The server checks to see if the RPD entry (see Figure 4.3) corresponding to the updated page already has ROD entries corresponding to the updated object linked to it. If other objects belonging to the page are already present in the MOB, then the server increments the MOBPresent counter and the TotalMOBAccess counter. If other updated objects belonging to the page (present in the MOB due to previous transfers from the client) are not present in the MOB, then the server increments only the TotalMOBAccess counter. The ratio of MOBPresent counter over the TotalMOBAccess counter is known as the MOB absorption rate. For MOB absorption rate values between 25 and 35 percent and disk utilization values between 75 and 85 percent there was no appreciable difference in the overall system throughput. If an MOB absorption threshold value smaller than 30% is chosen, then this aggressively favors sending pages to the clients and full MOB

absorption capability is not being utilized. That is, the server receives updated pages from the clients and, therefore, the MOB is not able to batch the installation of updated objects to its full potential. Similarly, if the MOB absorption threshold value that is larger than 30% is chosen, then clients return updated objects, and due to low MOB absorption rate, separate installation reads are performed for each of updated objects. Similarly, if the disk utilization threshold that is lower than 80% is chosen, then the ability to perform installation reads in the background is not being fully utilized. This, in turn, means that the MOB's absorption capability is not being fully utilized. If a higher disk utilization threshold is used then this increases the possibility that object groups are transferred between the server and the clients. This, in turn, increases the probability that installation reads are interfering with normal client read requests.

Other heuristics using other system parameters (such as CPU and network utilization) are possible but these are not considered in this dissertation.

- **Server-to-Client Data Transfer Rule Not Satisfied:** The server utilizes the client hint because the client knows more about its data access pattern and, thus, it can make a better decision. If the client requests a page, then the server returns the page to the client. If the client requests an object group, then it also sends the size of the object group. The server dynamically makes logical partitions of the page into n equally-sized sub-segments whose size is approximately equal to the size of the object group requested by the client [LAC+96]. The server then returns to the client the sub-segment in which the requested object resides (the object might not necessarily be the first object in the sub-segment). If the requested object's size turns out to be larger than the object group size hint provided by the client, then the server returns the entire object to

the client. If the size of the object is larger than the size of the page, then a special large object handling mechanism is required. The details of handling such large objects are beyond the scope of this dissertation. The server also sends a hint to the client informing whether its read buffer miss ratio is greater than 30%. The details of the read buffer miss ratio calculation have already been described above. The server sends this hint along with the data to the client. If the read buffer cache miss rate is high, then the server considers its read buffers to be contended. The client, in turn, uses this hint to determine whether it should request pages or objects from the server.

3. **Client Receives Object Group:** If the client receives an object group, it registers the objects into the resident object table and loads each of the objects in the object buffer. Subsequently, the client takes the following actions:

   - **Client-to-Server Data Transfer Rule Number 1:** If the BadAccessLocality/ClientCacheMiss ratio (also known as PageAccessFactor) is greater than 30% or the spatial locality is greater than 30%, then request pages from the server. Otherwise, the client requests object groups from the server.

   - **Rule Analysis and Alternatives:** It is beneficial to request a page if the spatial locality is greater than 30% because this reduces the object group forming overhead at the server and the object group disassembly overhead at the client. Furthermore, requesting pages also helps the client to overcome the inefficiency associated with the server object group forming mechanism. The client relies on its dual buffering mechanism to retain only useful objects from badly clustered pages. If the client requested object groups when the spatial locality is greater than 30%, then the client has to make more data requests to the server to overcome server grouping inefficiency. However, when the spatial locality is lower than 30% and

the PageAccessFactor is less than 30%, it is desirable to request object groups because of the CPU and network overhead associated with transferring badly clustered pages. Furthermore, storing badly clustered pages in the client buffer reduces the buffer utilization at the client.

The notion of PageAccessFactor represents the access locality characteristics of the client workload. If PageAccessFactor is greater than 30%, then it represents bad access locality. When the access locality is bad, then even if the spatial locality is less than 30%, it is desirable for the client to request pages because the server-based object group forming mechanism constructs object groups consisting of contiguously placed objects. Bad access locality means that the objects are accessed by the client in a non-contiguous manner. Therefore, there is an inherent mismatch between bad access locality and the server-based contiguous object grouping mechanism, and requesting objects results in a greater number of client cache misses and subsequent data requests to the server.

- **Threshold Calculations:** Both the PageAccessFactor threshold and the spatial locality threshold were empirically determined. In order to calculate PageAccessFactor, during each cache miss at the client, a check is made to see whether any other objects belonging to the same page are present in the client cache. That is, it is determined whether the client has previously accessed a different part of this page. This is determined by checking to see if there are any other resident object descriptors (RODs) corresponding to the objects of this page that are present at the client. As described in Figure 4.3, the client maintains RPT and ROT tables containing RPD and ROD entries corresponding to the pages and objects residing in the client cache, respectively. The RODs corresponding to the objects belonging to the same page are linked together as a list and there is a pointer from the RPD corresponding to the page

to the head of the ROD list. Thus, during a client cache miss, a check is made to see whether or not the pointer from the RPD to its corresponding ROD list is set. The client also maintains a badAccessLocality global counter and a clientCacheMiss counter. During a client cache miss the clientCacheMiss counter is incremented. The badAccessLocality counter is incremented if there are other objects corresponding to the page present in the client cache (the ROD list pointer from RPD is set to null). If the badAccessLocality/clientCacheMiss ratio (known as PageAccessFactor) is greater than 30 percent, then the client's page access locality is considered to be bad. This ratio represents the number of client cache misses that could have been avoided if the page corresponding to the requested object is present in the client cache. For values between 0.25 and 0.35, the overall performance did not change appreciably. If a PageAccessFactor threshold that is larger than 0.30 is used, then the client continues to request object groups even though the access locality is bad. This, in turn, results in a higher number of client cache misses, and, thus, an increased number of data requests to the server. If a PageAccessFactor threshold value that is smaller than 0.30 is used then this prevents the client from taking advantage of receiving object groups from the server when the spatial locality is poor. Thus, the client does not take advantage of lower network and CPU costs.

If the server read buffer is contended then the client decides to request pages when the PageAccessFactor is greater than 20 percent because in this case there is a greater probability that a client cache miss also results in a server cache miss and as discussed above, it is not desirable to request object groups when access locality is bad. That is, client access locality has to be much better in the server read buffer busy case than in the server read buffer not busy case in order for the client to continue requesting object groups from the

server. For values between 0.15 and 0.25, the overall performance did not change appreciably.

PageSpatialLocality is the ratio between the current object group size maintained by the client and the size of the page. For PageSpatialLocality values between 0.25 and 0.35, the overall system throughput does not change appreciably. If a PageSpatialLocality threshold value that is greater than 30% is used, then this favors the client requesting object groups from the server. This, in turn, makes it necessary for the server object grouping mechanism to be accurate in order to make transferring object groups advantageous. If a PageSpatialLocality threshold value that is lower than 30% is used, then this aggressively favors requesting pages, and in cases where the spatial locality is poor, the low CPU and network overhead benefits are not realized by the server and the clients.

- **Client-to-Server Data Transfer Rule Number 1 is Not Satisfied:** When the conditions of this rule are not satisfied then the client decides to request object groups from the server, and it also sends the object group size hint to the server along with its request. The client determines the object group size by keeping track of the amount of data (size of the objects in bytes) that it has accessed in the previously received object groups [LAC+96]. If a large portion of the previously received object groups has been accessed, then there is high spatial locality. The object group size is dynamically increased if the data access pattern matches the data clustering pattern on disk, and decreased otherwise.
  - **Object Group Forming Hint Rule:** If use/fetch ratio is less than a threshold of 0.3 then the object group size is decreased by 2% of the page size; otherwise, the object group size is increased by 2% of the page size.
  - **Rule Analysis and Alternatives:** The *use/fetch* ratio helps to determine whether the object group size is too small or too

large. *Fetch* is the size of the data in the object group received by the client, and *use* is the amount of data (in bytes) that has been used for the first time by the client after the data has been loaded into the client cache. This rule allows the client to dynamically increase the group size if it is using more than 30% of the previously received object groups, and decrease the object group size if it is using less than 30% of the previously received object groups.

- **Threshold Calculations:** The increment value and the use/fetch ratio threshold value were determined empirically. For increment size values between 1 and 3 percent, the overall performance did not change appreciably. The upper limit of the group size is the page size, and the lower limit is the increment itself (2 percent of the page size). If the increment size is larger than 2% of the page size then this results in the transfer and caching of more unused objects in the client cache. If the increment size is smaller than 2% of the page size, then this results in the client making multiple requests to the server to get the relevant data loaded into its cache.

  When an object group of size N bytes arrives, the client recalculates the *fetch* and *use* parameter values. For threshold values between 0.25 and 0.35, the overall system throughput does not change appreciably. If the threshold value is less than 0.3 then the client is not able to reduce the object group size, and this results in large object groups being transfered even when the spatial locality is poor. Similarly, if the threshold value is greater than 0.3, then the client is not able to increase the object group size quickly enough, and this results in the client sending multiple data requests to the server.

4. **Client Receives Page:** If the client receives a page then it registers it in the resident page table, and puts it into its page buffer. The page stays

in the client page buffer as long as there is no client buffer contention and the page is well clustered. Otherwise, the client flushes the page and retains only the objects that have been already used by moving them to the object buffer [KK94]. The client dual buffer management details are presented in Section 3.2. The client enforces the following rule to switch from requesting pages to requesting object groups.

- **Client-to-Server Data Transfer Rule Number 2:** If the page spatial locality is less than 30% and if the PageAccessFactor is greater than 30% then the client switches over to requesting object groups. Otherwise, the client continues to request pages from the server.

- **Rule Analysis and Alternatives:** If the page spatial locality is less than 30% then the clients decide to request object groups because it is not desirable to cache badly clustered pages in the client cache. Furthermore, higher network and CPU overhead is incurred when badly clustered pages are transferred between the server and the clients. However, in addition to the page spatial locality, the clients also check the access locality (PageAccessFactor) because, as described above, it is not desirable to transfer object groups when the access locality is bad (PageAccessFactor is greater than 30 %) because this results in a higher number of client cache misses.

- **Threshold Calculations:** Even though pages are received from the server, the client still keeps track of the desired group size. The group size value is still calculated in the same manner as when the client requests object groups. For group size threshold values between 25% and 35%, the overall system throughput does not change appreciably. If a larger threshold value had been chosen, then the client would prefer object groups much more aggressively, and would pay the penalty of higher client cache misses due to server grouping inefficiency. Similarly, if a smaller threshold value

than 30% had been chosen, then the client would request pages with low spatial locality and would incur lower client buffer utilization and higher network and CPU overheads. When the client is manipulating pages, it calculates the PageAccessFactor as a ratio of object group size over the range of objects accessed in a page. The range of objects represents the lowest byte offset object and the highest byte offset object that have been accessed from the beginning of the page. The object group range and the page access factor are calculated every time the client ejects a page from the client dual buffer and copies the useful accessed objects from the evicted page into the dual buffer. Since information about the location of all the useful objects (residing in the page to be flushed) is accessed when the objects are copied from the page buffer into the object buffer, the PageAccessFactor ratio is calculated when a page is flushed from the page buffer. The PageAccessFactor ratio helps to determine whether the access locality is good or bad. If the PageAccessFactor threshold is higher than 30%, then the clients aggressively request pages and, therefore, pages with low spatial locality are transferred from the server and cached at the client. However, if the PageAccessFactor threshold is lower than 30%, then the clients would request object groups and incur higher client cache misses due to the inefficiency in the server object grouping mechanism.

5. **Client Returning Updated Data:** When a client performs an update, it can return either an updated page or updated objects. If the server has passed a hint indicating that the server buffer is busy to the client, and the updated page exists in the client cache, then the client returns the updated page to the server. Otherwise, the client returns updated objects of the page to the server. The client does not want to return updated objects when the server buffer is contended because it wants to reduce the installation read interference with normal client data request reads that are performed at the server.

During absence of server buffer contention, the clients prefer to return updated objects because installation reads will not be a problem at the server. Also, returning updated objects reduces network overhead and also helps to increase the server MOB absorption rate by batching the installation of many updated objects to their corresponding pages. Section 3.2 describes the details of MOB operation and how it reduces the number of installation reads and writes.

6. **Server Receiving Updated Data:** After receiving the updated objects/page from the client, the server loads them into its modified buffer, and then flushes them to disk in the background.

The ratio values that are used in the rules above have been calculated once every 100 milliseconds. Moreover, these values are exponentially forgotten to prevent thrashing behavior. For example, once the new disk utilization value is calculated, it is added to the existing disk utilization value and the sum is divided by 2 (exponential forgetting) [LAC+96]. It was observed that, at current hardware speeds, re-calculating the three values every 100 milliseconds instead of every 1 second or every 1 millisecond provides a good balance between accuracy and monitoring overhead.

In conclusion, the following features are unique to the adaptive data transfer mechanism in comparison to the previous data transfer approaches:

- **Adaptive Server-to-Client Data Transfer:** This is the first dynamic data transfer mechanism to utilize an adaptive data transfer approach in both server-to-client, and client-to-server directions. Previous data transfer approaches did not switch between sending pages or groups of objects from the server to the client [LAC+96, DFMV90].

- **Adaptive Client-to-Server Data Transfer:** The adaptive client to server data transfer mechanism proposed here takes server buffer contention level, client buffer management, and network cost into account while deciding whether to return updated pages or objects to the server.

The previous client to server data transfer approaches [Ghe95, OS94a] did not take all of these factors into account.

- **Support for Varying Object and Page Sizes:** The previous object group forming mechanism [LAC+96] did not take varying object and page sizes into account, whereas the object group forming mechanism proposed herein handles varying object and page sizes.

- **Support for Varying Access Locality:** The previous object group forming mechanism [LAC+96] did not account for non-contiguous access to a page because the client only kept track of the number of objects that have been accessed in the client cache, and it was not concerned about the access locality characteristics. Therefore, it did not take the notion of access locality into account. The adaptive data transfer mechanism presented here takes varying access localities into account, and it uses this information to switch between requesting pages and object groups.

4.4 Performance Results Overview

The server-to-client data transfer and client-to-server data transfer mechanisms are evaluated as part of the integrated performance study in Chapter 8. The simulation-based integrated performance study compares the new adaptive data transfer approach with sending either only pages or objects among the server and the clients. The key results of the data transfer study are:

- Adaptive data transfer approach is more robust with respect to performance than page or object-based data transfer approaches. Thus, each of the data transfer rules in this chapter have been validated.

- It is difficult to have an efficient server-based object grouping mechanisms when the data access locality is bad because the server-based object grouping mechanisms form object groups that contain contiguously placed objects on the disk.

- It is desirable to send pages from the server to the clients when the server buffers are contended because there is a higher chance of a miss at the client cache being also a miss at the server cache.
- It is desirable to send updated objects to the server when the server buffers are not heavily contented and it is desirable to send updated pages to the server when its buffers are heavily contended. Taking server buffer contention into account helps to optimize the installation I/O overhead.
- The dual buffer at the client can be used efficiently in conjunction with the adaptive data transfer mechanism to improve overall system performance.
- Returning updated pages to the server when the server buffers are not contended negates the benefits of hardware pointer swizzling for small transaction sizes (200 objects accessed) and for write probabilities greater than 2%.

Chapter 5

Cache Consistency

Adaptive Callback Locking (ACBL) [CFZ94] and Adaptive Optimistic Concurrency Control (AOCC) [AGLM95] are currently the two prominent client-server ODBMS cache consistency algorithms. AOCC is an optimistic algorithm which, for certain workloads, has better performance than ACBL, but is susceptible to high abort rates. ACBL is a pessimistic algorithm that has a lower abort rate than AOCC, but its performance trails AOCC's due to higher message processing and blocking overheads. This chapter presents the Asynchronous Avoidance-based Cache Consistency (AACC) algorithm. AACC builds upon the strengths of AOCC and ACBL while avoiding their weaknesses. It is an adaptive algorithm because the clients and the server can dynamically adapt between sending synchronous, asynchronous or deferred lock messages. Furthermore, it can be efficiently used by both page and object server architectures.

This chapter first briefly describes the key factors that affect the performance of a cache consistency algorithm. It then describes how ACBL and AOCC address these problems, and motivates the design of AACC by discussing how AACC tackles the same issues. This is followed by the presentation of the AACC algorithm for page server architectures. It finally extends the AACC algorithm so that it can also be used by object and hybrid server architectures. For an overview of basic client-server cache consistency concepts the reader is referred to Chapter 2.

5.1 Cache Consistency Overheads

The four key factors which determine the performance of cache consistency algorithms are the write lock message transmission overhead, write lock message blocking overhead, abort processing overhead and lock conflict blocking overhead. The following is a description of each of these overheads:

- **Write Lock Message Transmission Overhead:** This is the CPU processing cost associated with sending and receiving explicit locking related messages at the clients and the server. This is predominant in ACBL.

- **Write Lock Message Blocking Overhead:** This overhead is encountered when the clients send synchronous lock escalation messages to the server. The client remains blocked till the server returns a response to the client. This is present only in ACBL.

- **Lock Conflict Blocking Overhead:** This overhead is incurred when a transaction blocks due to a read-write or write-write locking conflict. It is present in ACBL and AACC.

- **Abort Processing Overhead:** When a transaction aborts due to a deadlock or due to a stale cache access, the aborted transaction has to be re-executed and there is an associated cost. This overhead is present in all of the cache consistency algorithms, but it is predominant in AOCC.

The proportion of each of these overheads vary in each of the cache consistency algorithms as the workload and system configurations change. This chapter presents four cache consistency scenarios to describe these costs and how the algorithms deal with them. These scenarios help to intuitively describe how AACC tries to minimize each of these overheads.

5.2 Adaptive Callback Locking (ACBL)

ACBL is a synchronous, avoidance-based cache consistency algorithm [CFZ94]. Clients cache both data and read locks across transaction boundaries, but they need to obtain write permission from the server before they can proceed with write operations. ACBL can dynamically acquire either page or object-level locks, and thus, it is an adaptive version of the page-level CBL algorithm [FC94]. Clients try to acquire page-level write locks; failing that, they try to acquire object-level write locks on shared pages. If the page is cached at other clients, the server sends callback messages to these clients asking them to downgrade or relinquish their locks. ACBL ensures that transactions never access stale data and, therefore, never have stale cache aborts. However, one can encounter deadlock related aborts. The following four scenarios (Figure 5.1) are used to show the operation of ACBL. For simplicity, these scenarios deal with only two clients, but the discussion is valid for n clients.

- **Scenario 1:** Assume that page 1 is only cached at client 1 that has a read lock on page 1. Client 1 wants to update object 1 on page 1 and, therefore, it sends a message to the server to obtain a write lock for page 1. Client 1 blocks until it gets a response from the server. Since there is no other client that caches page 1, the server immediately grants the write lock. Thus, even if a page is not cached elsewhere, in ACBL the clients send lock escalation messages to the server and block until they get a response from the server. Thus, ACBL encounters write lock message transmission overhead.

- **Scenario 2:** Client 1 wants to update object 1 on page 2 which is also present at client 2 due to inter-transaction caching; however, it is not being actively used at client 2. Both clients hold a read lock on the page. Client 1 sends a lock escalation message to the server and blocks until it gets a reply. The server, in turn, sends a callback message to client 2. In general, the goal of a callback message is to invalidate the object/page cached at remote clients so that the lock requesting client can proceed with its write operation. In this scenario, since client 2 is not using page

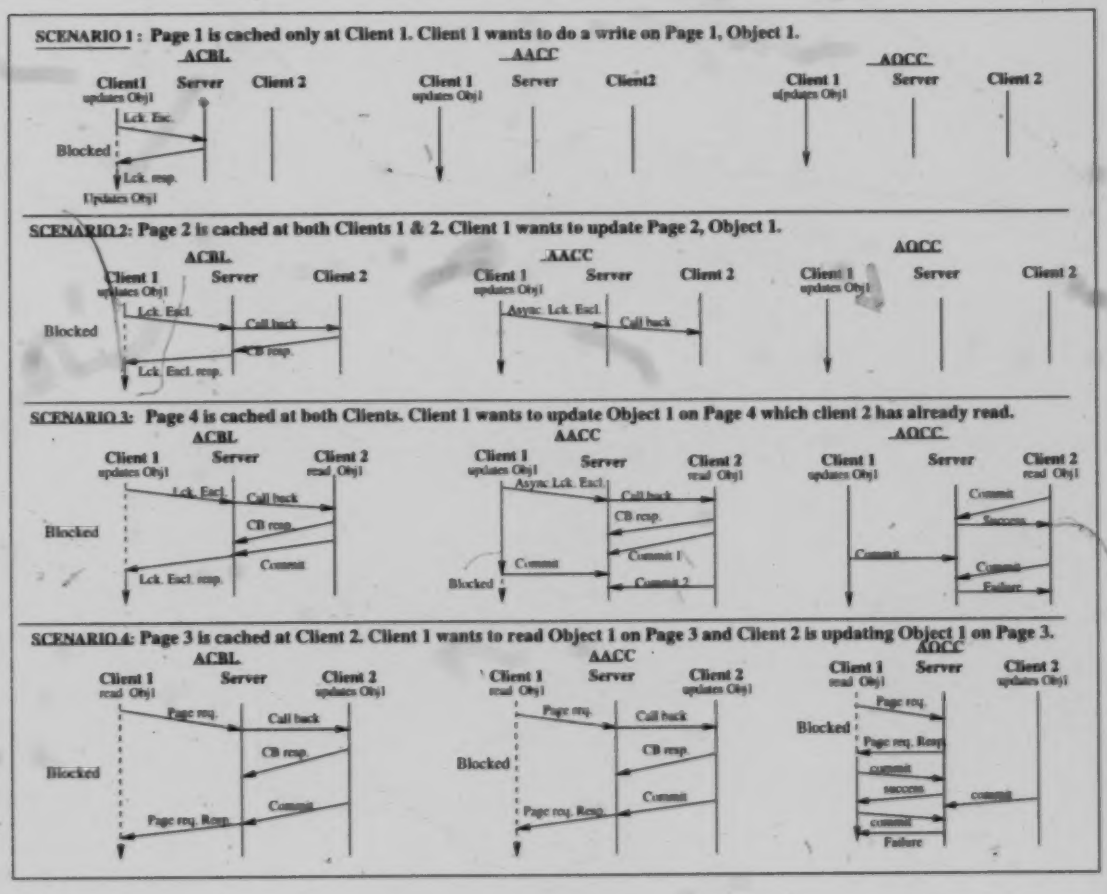

SCENARIO 1 : Page 1 is cached only at Client 1. Client 1 wants to do a write on Page 1, Object 1.
ACBL
AACC
AOCC
Client1
Server
Client 2
Client2
updates Obj1
Lck. Esc.
Blocked
Lck. resp.
Updates Obj1
SCENARIO 2: Page 2 is cached at both Clients 1 & 2. Client 1 wants to update Page 2, Object 1.
Lck. Excl.
Call back
CB resp.
Lck. Excl. resp.
Async. Lck. Excl.
SCENARIO 3: Page 4 is cached at both Clients. Client 1 wants to update Object 1 on Page 4 which client 2 has already read.
read Obj1
Commit
Commit 1
Commit 2
Success
Failure
SCENARIO 4: Page 3 is cached at Client 2. Client 1 wants to read Object 1 on Page 3 and Client 2 is updating Object 1 on Page 3.
Page req.
Page req. Resp.

Figure 5.1: Cache Consistency Scenarios

2, it invalidates the page from its cache and sends a callback reply to the server. The server then sends a response to client 1, granting it an exclusive lock on page 2. Thus, when a page is cached at multiple clients, in addition to the round trip message between the lock requesting client and the server, there are round trip callback messages between the server and all of the other clients where the page is cached. The initial lock requesting client blocks until all of these messages are processed (even when the desired page and object are not used elsewhere). Thus, in ACBL, the clients encounter write lock message blocking overhead.

- **Scenario 3:** Page 4 is shared by both clients 1 and 2. Client 1 wants to update object 1 on page 4 and client 2 has already read the object. Client 1 sends a lock escalation message to the server which then sends a callback message to client 2. Client 2 indicates that it cannot comply with the request. Client 1 stays blocked until client 2 commits and releases the page. Thus, in ACBL the client transactions remain blocked during locking conflicts to avoid transaction aborts and, hence, they encounter lock conflict blocking overhead.

- **Scenario 4:** Client 2 holds an exclusive lock on page 3 and is updating object 1. Client 1 wants to read object 1 on page 3 and it sends a message to the server to obtain page 3. The server sends a callback message to client 2 which responds by indicating that it is updating object 1 on page 3. Client 1 remains blocked until client 2 commits. Thus, read operations remain blocked until the appropriate lock is obtained from the server to prevent a transaction abort. This scenario highlights that ACBL has a lower abort rate than AOCC, and this is considered to be a strength of ACBL.

5.3 Adaptive Optimistic Concurrency Control (AOCC)

AOCC is a deferred, detection-based cache consistency algorithm. In AOCC, clients implicitly obtain read permissions on cached data, but if they subsequently update cached data, they defer all of their write notification messages until commit time. AOCC does not prevent the access of stale data by clients. The updates of a committed transaction result in corresponding invalidations being sent to other affected clients. These invalidations are piggybacked (not an explicit message) on other messages. Explicit messages incur the entire network protocol stack overhead, whereas, the piggybacking helps to amortize the protocol stack overhead by batching and transmitting multiple high level messages as a single message. If the client that receives an object invalidation has accessed the corresponding object, then it performs a stale cache abort. Since this is an optimistic algorithm and no locking is involved, clients do not encounter read/write or write/write blocking and, therefore, deadlocks do not occur in AOCC. However, in addition to stale cache aborts, it is susceptible to starvation. That is, a client transaction could repeatedly abort and never be able to commit.

In AOCC, the server has to perform commit time validation on every object that has been accessed by a transaction. The server checks whether the client has accessed the most recently committed version of the object. This validation overhead is not present in ACBL since the algorithm ensures that clients do not access stale data. In AOCC, for each client the server maintains an invalidation queue that stores the list of committed updates of other clients that can potentially have an impact on client i. The invalidation queue is used by the server while performing commit time validation.

The same scenarios as before (Figure 5.1) are used to analyze AOCC:

- **Scenario 1:** Client 1 wants to update object 1 on page 1 and it is the only client caching that page. It does not send any lock escalation messages to the server for this update; it simply goes ahead and performs

its update on object 1 on page 1. The server is notified about this update by the client during its commit operation. Thus, in AOCC, there is no write lock message transmission overhead.

- **Scenario 2:** Client 1 wants to update object 1 on page 2 which is also cached at client 2. Client 1 does not send any lock escalation message to the server; it goes ahead and performs its update on the object. Client 1 informs the server about the update during its commit operation. Therefore, the server does not send any callback messages to client 2, but piggybacks an invalidation message to client 2 because the data is cached at client 2. Thus, there is no write lock message blocking overhead in AOCC.
- **Scenario 3:** Client 1 wants to update object 1 on page 4. This page is cached at both clients 1 and 2, and the latter has already read object 1 on page 4. Client 1 does not send any lock escalation messages to the server for the update; it informs the server during its commit operation. If, at commit time, the server detects that client 1 transaction has not yet committed, then client 2 transaction commits (sneaks through), followed by the client 1 transaction. If client 1 has committed ahead of client 2, then the client 2 transaction aborts. Thus client transactions never block in AOCC due to a locking conflict.
- **Scenario 4:** Page 3 is cached at client 2 and object 1 on this page has been updated by this client. Client 1 wants to read the same object. Client 1 goes ahead and gets page 3 from the server, and it accesses object 1. If client 1 transaction commits before client 2, then it sneaks through and successfully commits. If client 2 commits before client 1, then client 1 aborts. In AOCC, the absence of transaction blocking during a locking conflict can potentially lead to stale cache transaction abort.

In ACBL, a read/write conflict always results in blocking one of the transactions; in AOCC, the reading transaction can successfully commit (sneak through) if it reaches the commit point first, and the reading transaction

aborts if the writing transaction commits first. This causes the blocking rate of ACBL to be higher than the abort rate of AOCC, but the abort rate of AOCC is higher than the abort rate of ACBL. In AOCC, when a transaction aborts, the client simply copies the undo logs that are maintained in its memory and restarts the transaction. This, in turn, speeds up abort processing as, for most non-conflicting objects, the client does not have to go to the server again to obtain the necessary pages.

5.4 AACC Algorithm

This section first describes the five novel features of AACC and then describes the AACC algorithm in detail. The five key features are:

- **Shared/Private Locks:** AACC introduces the notion of private and shared page-level lock messages. In AACC, pages can be locked in *private-read*, *shared-read* and *page-write* modes, and objects can be locked in *read* and *write* modes. The transition between these lock modes is described below in the detailed description of AACC.

  When the server is returning a page to the client, the server also informs the client whether or not the page is cached anywhere else. If the page is cached in another client's cache, then the server sends the page to the requesting client in shared read lock mode, or else it sends the page in private read lock mode. If a client has a page in private read lock mode, then it piggybacks the write lock requests for that page to the server with other messages that have to be sent to the server to reduce its message transmission overhead.

- **Asynchronous Locking Messages:** AACC uses asynchronous lock escalation messages on pages that reside in the caches of multiple clients. This, in turn, helps to avoid the message blocking overhead that is present in algorithms that use synchronous locking messages. Moreover, in comparison to algorithms using deferred locking messages, the use of asynchronous locking messages reduces deadlocks by informing

the server sooner about client lock escalations (this is described further in Section 5.6).

- **Avoidance-Based:** AACC is an avoidance-based algorithm and therefore, it never accesses stale data. Hence, AACC never encounters stale cache aborts.

- **Piggybacking of Callback Locking Messages:** When the server receives a lock escalation message for a page from a client, and this page is cached at a second client, then the server issues a lock callback message for the page to the second client. In AACC, the second client piggybacks its lock callback response to the server if there is no explicit object-level locking conflict. This, in turn, helps AACC to reduce the lock message transmission overhead.

- **Deadlock Avoidance:** Since AACC uses asynchronous lock escalation messages, there is a greater probability of encountering a deadlock than in algorithms that use synchronous lock escalation messages like in ACBL. Therefore, AACC employs two deadlock avoidance optimizations that help it to reduce its deadlock rate. These deadlock optimizations are presented in Section 5.6.

The performance implications of each of these features are evaluated in Chapter 8.

5.5 Intuitive Description of AACC

This section uses the four scenarios of Figure 5.1 to discuss how AACC handles the four previously discussed overheads associated with client-server DBMS cache consistency algorithms.

- **Scenario 1:** In ACBL, the client sends an explicit lock escalation message when it wants to update object 1 on page 1. In AOCC the client defers informing the server about the write operation until commit time.

Therefore, it does not encounter the message passing overhead that is present in ACBL. Normally, a delay in informing the server about updates until commit time increases the probability that another client has read the same object. This, in turn, increases the probability of an abort. However, since this page is cached only at a single client, the chances of conflicts are remote. AACC tries to capitalize on this insight by introducing the notion of private and shared pages. That is, when a server sends a page to the client, it also informs the client whether that page is cached elsewhere. If it is not, then the client piggybacks its write lock request instead of sending an explicit lock request. Therefore, as shown in Figure 5.1 scenario 1, if Client 1 wants to update object 1 on page 1, that is cached only at client 1 in private-read lock mode, then the client goes ahead with the update without sending an explicit write lock message. The client informs the server about this update by piggybacking the lock escalation message on a subsequent message to the server. The lock message is piggybacked instead of being deferred until commit time, to reduce the risk of a read/write conflict for the particular object. Thus, the notion of shared and private page read locks reduces the write lock message traffic in AACC.

- **Scenario 2:** This scenario shows the message blocking overhead that is present in ACBL since client 1 waits until its write lock request for object 1 on page 2 is granted by the server. The server, in turn, issues a lock callback message to client 2, and only grants the request to client 1 after hearing from client 2. Once again AOCC defers the write lock request until commit time, but increases the probability of a locking conflict and a subsequent transaction abort. AACC does not want to send synchronous locking messages as in ACBL, but at the same time does not want to defer the lock message to the server (informing about the update) until commit time and thus, increase the probability of an abort. Hence, in AACC, when Client 1 wants to update object 1 on page 2 which is cached at both clients 1 and 2 in shared-read lock mode,

Client 1 sends an asynchronous lock escalation message to the server and continues without blocking. The server, in turn, forwards this message to client 2, which invalidates page 2, but informs the server about this invalidation by piggybacking the information on a subsequent message. Thus, asynchronous locking messages have been introduced in AACC to reduce the write lock message blocking overhead. The sending of an asynchronous lock message does not delay the informing of the update to the server and it thus, reduces the abort probability. The piggybacking of the callback response from client 2 also reduces the lock messaging overhead.

- **Scenario 3:** As shown in Figure 5.1 scenario 3, in ACBL, client 1 remains blocked until the client 2 transaction commits. In AOCC, client 1 transaction does not block and, therefore, there is a greater probability of a transaction abort. AACC does not want to increase the probability of a transaction abort, but at the same time, it wants to reduce the time a transaction remains blocked due to a locking conflict. As shown in Figure 5.1 scenario 3 for AACC, client 1 does not block at the point there is a possibility of a read/write conflict but it instead blocks at commit time. Therefore, instead of remaining blocked until client 2 transaction commits, client 1 is able to continue with its transaction execution from the point of the conflict until its transaction commit point, and is thus able to increase the overall system throughput. For example, in scenario 3, client 1 sends an asynchronous message to the server indicating its update. The server then forwards this message to client 2. Client 2 notices that there is a conflict and sends an explicit response to the server. The server then performs deadlock processing and notes that client 1 can only commit after client 2 has committed in order to prevent stale cache aborts. Therefore, client 1 can go ahead with its commit if client 2 commits at commit point 1 but client 1 blocks if client 2 commits at commit point 2.

- **Scenario 4:** As shown in scenario 4 of Figure 5.1, ACBL blocks when there is a lock conflict, whereas AOCC increases its probability for a transaction abort by not blocking one of the conflicting transactions. A high number of aborts are not acceptable in many interactive transaction domains, and in environments where the cost of abort processing is high. Similar to ACBL, AACC also blocks when there is an explicit locking conflict. In scenario 4, Client 1 wants to read object 1 on page 3, which is present only at client 2. Moreover, client 2 holds an exclusive page level lock on page 3 and it is also updating object 1 on page 3. Upon receiving the page 3 read request from client 1, the server sends a callback message to client 2. Since client 2 is using the object, it sends a negative response to the server and thus client 1 blocks until client 2 does a commit. Just like the use of asynchronous lock escalation messages, instead of deferred lock escalation messages reduces the abort probability (as described above), the use of synchronous lock escalation messages reduces the probability of an abort even further. In order to ensure that it has as low an abort rate as ACBL, AACC incorporates two deadlock avoidance optimizations that are described in Section 5.5. Furthermore, since ACBL and AACC are avoidance-based algorithms, they do not allow for the presence of stale cache data in the client cache.

5.6 AACC Detailed Description

The following is a detailed description of AACC.

- **Data Request:** When a client wants to access an object whose page is not in its cache, it sends a page request to the server. When the server receives the request, it checks to see whether the page is cached at other clients.
  - If the page is not cached anywhere else, it returns the page to the client in *private-read* mode.

- If the page is cached at another client in *private-read* mode, then the page is returned to the requesting client in *shared-read* mode. The server also informs, via a piggyback message, the client holding the page in *private-read* mode to change the page lock to *shared-read* mode. The inherent message delay may cause situations where one client has the page in *private-read* mode and other clients have the same page in *shared-read* mode.
- If the page is cached elsewhere in *shared-read* mode, then the server returns the page to the client in *shared-read* mode.
- If the page is cached at another client in *page- write* mode, then the server issues a callback message to the remote client indicating the object and the page that is being requested. Upon receiving the callback, the remote client checks to see whether it is using the particular object. If not, it changes the page lock to *shared-read* and returns the object identifiers of the objects on that page that have been updated. If it is using the requested object, it informs the server that it cannot satisfy the request.
- Upon receiving a positive callback response, the server marks off the objects that are updated at the remote client and sends the page to the requesting client. If the server receives a negative callback response, it blocks the requesting client until the client that holds the write lock commits.

- **Updates on Private-Read Locked Pages:** When a client is performing an update on a *private-read* locked page, the client changes the page lock mode to *page-write*. The client then informs the server about this update by piggybacking the information on a subsequent message. Upon receiving the piggybacked message regarding the update and the lock escalation to the *private-read* locked page, the server does the following:
  - If the page is residing at other clients in *shared-read* lock mode, then the server sends an invalidation message to the affected clients. The invalidation message requests the clients to purge the object and/or

page from their caches. The server also informs the client that has performed the update to change its page lock for the updated page from *page-write* to *shared-read* if other clients are using the page but not that object.

  - If the page is not present at other clients or has been successfully invalidated, then the server updates its lock tables to indicate that the client has a *page-write* lock for the page.

- **Updates on Shared-Read Locked Page:** When a client is performing an update on a *shared-read* locked page, it sends an asynchronous lock escalation message to the server and continues with its processing. When the server receives this message, it sends callback messages (indicating both the object and the page) to the other clients that have cached this page.

  - If the client that receives the callback message is not using the page, it simply invalidates it, and informs the server via a piggybacked message.
  - If the client is using the page but not the object, then it invalidates the object and informs the server via a piggybacked message.
  - If the client is using the object, then it sends a callback response indicating that there is a conflict.

- **Callback Processing:** When the server receives a callback response indicating that there is a conflict, it performs deadlock detection processing, and if there are no deadlocks, the client that has performed the initial update cannot commit before the client that is reading the object. Here, the server deadlock detection processing involves a check to see whether clients have updated objects that have been read by other clients. For example, if client 1 has updated an object read by client 2 and client 2 has updated an object read by client 1, then neither of these clients can commit their respective transactions and the server randomly aborts one of the conflicting transactions. If the server receives

piggybacked callback responses from all the relevant clients indicating that they have invalidated the page, it sends an asynchronous message asking the client updating the initial page to upgrade its page lock from *shared-read* to *page-write* mode.

- **Commit Processing:** At commit time, the client sends the logs to the server. The client also piggybacks messages informing the server of updates to *private-read* locked pages. If a client has performed updates to a *private-read* locked page, and this is being piggybacked on the commit message, then the server checks to make sure that no other client has that page in its cache in *shared-read* mode; and if another client does have that page, the server sends a callback message to that client. The server only allows the commit to proceed after receiving replies to all the pending callback messages from the necessary clients. At commit time, the server checks to see whether the particular client can go ahead with its commit or whether it should remain blocked since it has updated an object that has been read by another client. The server also moves logs to persistent storage, and then informs the client that it can go ahead with the commit. The client changes *page-write* page locks to *private-read* locks, and *write* object locks to *read* locks. The client relinquishes the objects that have pending callback messages on them from the server. The client then informs the server about its lock de-escalations; the server updates its page and object level lock tables accordingly. It also activates the other client transactions that are waiting for this client to commit.

5.7 Deadlock Processing Analysis

Similar to ACBL, AACC is an avoidance-based algorithm; therefore, it does not encounter stale cache aborts, but it does encounter deadlock-related aborts. Read/write and write/write conflicts can lead to stale cache aborts, whereas read/write or write/write sharing across multiple objects is required in order

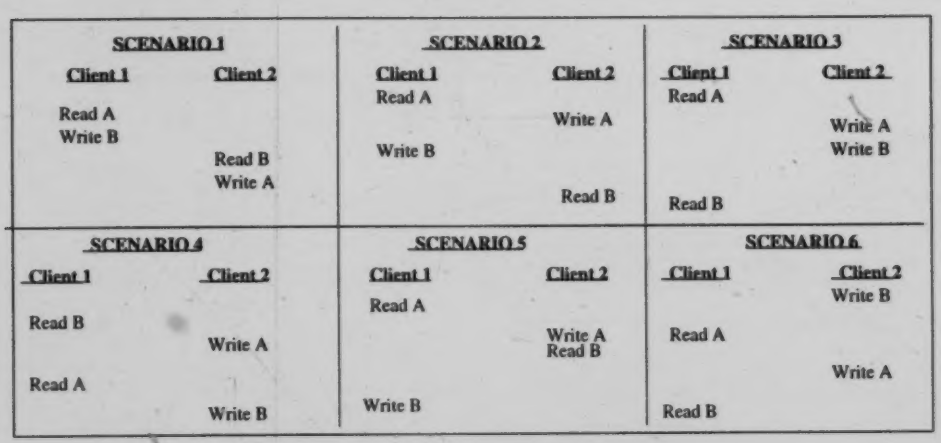


Figure 5.2: Deadlock Scenarios

for deadlock related aborts to occur. In most workloads, there is a low probability for the deadlock to occur because sharing is coincidental. The deadlock abort rates of ACBL and AACC are expected to be much lower than the stale cache abort rate of AOCC.

An important advantage of using asynchronous lock escalation messages is that it lowers the number of deadlock related aborts relative to what occurs with deferred lock escalation messages. Asynchronous lock escalation messages are sent right away by the client to the server, whereas deferred messages are delayed and sent at commit time. Scenarios 1 and 2 of Figure 5.2 describe the types of deadlock aborts that are avoided if one uses asynchronous lock escalation messages, but are possible if one uses deferred lock escalation messages. In scenario 1, an asynchronous lock escalation message prevents client 2 from reading object B, and this, in turn, prevents a deadlock. In scenario 2, an asynchronous message prevents client 2 from reading object B and this again prevents a deadlock. This is the main reason why the optimistic two phase locking (O2PL) avoidance-based family of cache consistency algorithms [FC94], which utilize deferred messages, face an increase in the deadlock rate as data contention increases. This high deadlock rate has discouraged client caching DBMSs from using the O2PL family of cache consistency algorithms [FC94].

In order to further reduce the AACC deadlock abort rate to the level of ACBL abort rate, the following two deadlock optimizations are used in AACC:

- **Sneak-Through Deadlock Optimization:** The notion of sneak-through has been used to avoid the type of deadlocks illustrated by scenario 3 in Figure 5.2. Sneak-through refers to the situation where a client has read an object that has been updated (but not yet committed) by another client, and the client that has read the object commits its transaction before the conflicting client's transaction commits, preventing it from accessing stale data. In scenario 3, Client 1 has read object A prior to that object's update by client 2. This scenario is possible since, in AACC, update operations never block at the time of the update even during the presence of conflicting read/write operations. The updating transaction only blocks if it reaches the commit point before the reading transaction. Therefore, client 2's update of object A will make client 2 block at commit time. If client 2 updates object B before client 1, then client 1 will normally block. In these situations, the server realizes that since client 1 is already causing client 2 to block due to its reading of object A, client 1 itself should not block on object B. Hence, the server averts a deadlock. The server maintains the information that client 1 is in sneak-through mode with respect to client 2. This sneak-through optimization helps AACC to avoid deadlocks, shown in scenario 6, which occur in ACBL.

- **Blocking Reversal Deadlock Optimization:** When the server detects a deadlock, it checks to see whether the deadlock is of the type depicted by scenario 4 in Figure 5.2. In this situation the server unblocks client 1 (which was blocking on object A) and instead blocks client 2 at commit time to avert a deadlock.

As the experimental results in Chapter 8 show, the deadlock abort rate in AACC is very similar to ACBL's. However, AACC still encounters deadlock scenario 5 (Figure 5.2) which is not encountered by ACBL because, in the

latter algorithm, client 2 is not allowed to write object A and, therefore, client 2 transaction remains blocked and it does not access object B.

5.8 Hybrid Granularity Concurrency Control

Since a hybrid server can transfer both pages and objects from the server to a client, the cache consistency algorithm for hybrid servers must be able to efficiently handle both page and object-level granularity. Efficient low-abort cache consistency/concurrency control algorithms have been proposed for page server client caching architectures [CFZ94]. However, similar low abort cache consistency/concurrency control algorithms are not available for object servers, because the following outstanding problems still need to be resolved:

- **High Messaging Overhead:** In object servers, lock escalation messages from the clients to the server, lock grant messages from the server to the clients, and callback messages from the server to the clients have to be sent at the object-level. A previous performance study has shown object-level messages to be a key scalability drawback of object servers [CFZ94].

- **Server memory overhead:** In client-server architectures, the server has to keep track of the data and locks present in client caches. Therefore, the server lock table size is dependent on client cache status. Due to inter-transaction caching of data and locks at the clients, the lock entries in the server lock table are not removed at the end of transactions, but instead can exist for long periods of time. With object-level lock table entries, the server lock table size could become very large and could potentially become an issue in certain low-end server configurations with a modest amount of memory. That is, managing information at the server strictly at the object-level is not a scalable option.

- **Lock processing overhead:** There is a processing cost associated with each locking/unlocking operation. It is desirable to perform locking at

coarser granularity since it reduces this lock processing overhead at both the server and the clients.

To circumvent these scalability issues, researchers have proposed efficient optimistic cache consistency algorithms for object servers, which, in turn, have the side effect of having high abort rates [LAC+96]. However, algorithms with a high abort rate are not desirable from a performance and usability standpoint for many workload and system configurations. In order to propose a high performance/low aborting cache consistency algorithm for hybrid servers, one has to solve the above mentioned object server cache consistency problems.

5.8.1 Concurrency Control for Hybrid Servers

In the past, cache consistency and data transfer mechanisms have been looked upon as being tightly coupled. That is, if objects are transfered between a server and clients, then concurrency control is also managed at the object-level. Similarly, if pages are transfered then concurrency control is primarily managed at the page-level, and, only in cases of page-level lock conflicts, are locks managed at the object-level. As shown in Figure 5.3, the current notion is that page servers can lock data at either page or object-level, but object servers can only lock data at the object-level. This dissertation introduces the

	Data Transferred And Caching	Locking Operations	Callback Operations
Page Servers	Page	Page/Object	Page/Object
Object Servers Current Status	Objects	Object	Object
Object Servers Dissertation Proposal	Objects	LLS/Object	LLS/Object

Figure 5.3: Coupling between Locking and Data Transfer

notion of a *Logical Lock Segment* (LLS) which decouples the data transfer and

concurrency control mechanisms for object servers.

Logical Lock Segment (LLS)

An LLS is a unit of locking which can map to a single object, a page, or a group of objects. The LLS concept allows even object servers to efficiently use AACC algorithm. When the server returns a group of objects to a client, it also informs the client about the corresponding LLS(s) for the object group. An object belongs to a single LLS. That is, LLSs cannot be overlapping. They can be of varying sizes, but the size of an LLS can only be changed by the server (using a background process) when no client is actively using it. The server can split an LLS into smaller LLSs or it can join adjacent LLSs into a larger LLS. The adaptive hybrid locking protocol for page servers [CFZ94] allows page servers to lock data at the page-level, and if there are locking conflicts at the page-level, then locking is performed at the object-level on pages that incur conflicts. Similarly, the hybrid server architecture locks data at the LLS level, and if there are locking conflicts at the LLS level, then locking is performed at the object-level for LLSs that incur conflicts.

In this dissertation, object groups only consist of contiguously placed objects (on disk). Thus, the LLS only contains objects that are contiguous and belong to the same page, and the size of all LLSs remains constant (equal to the page size). Therefore, the simple notion of an LLS allows an object server to lock data at the page-level.

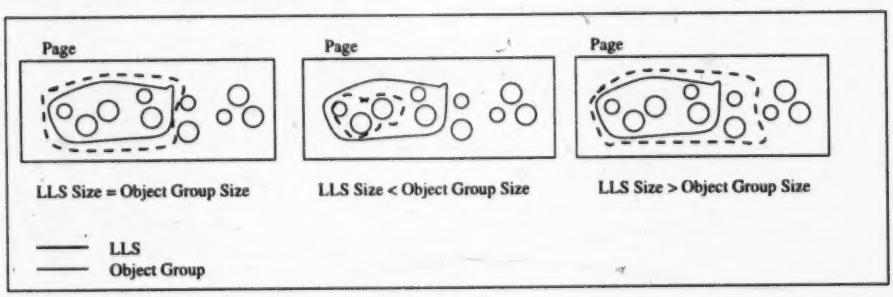


Figure 5.4: Logical Lock Segment

As shown in Figure 5.4, if the LLS size is larger than the size of the object

group then clients will lock objects that are not actually present in their caches. If the size of LLS is too small, then one is approaching object-level locking, and, therefore, one encounters the same types of problems as in object-level locking. The negative impact of a large LLS size is only felt if two clients want to lock the same LLS in conflicting modes. This will then force both clients to decrease the granularity of their locking from the LLS-level to the object-level.

Similar to the adaptive locking mechanism devised for adaptive page/object level locking in ACBL, the clients keep track of the LLS in an LLS table. Each object table entry in the client cache contains a pointer to its corresponding LLS entry in this table. Each entry in turn, contains links to the objects that belong to that LLS. The server maintains lock information primarily at the LLS-level. However, if there is an LLS-level lock conflict, then the server also maintains lock information at the object-level. Thus, the notion of an LLS minimizes lock escalation messages, reduces locking data structure memory overhead, and reduces the number of lock and unlock operations.

5.9 Performance Results Overview

The performance of AACC, ACBL and AOCC algorithms has been evaluated in Chapter 8. The simulation-based performance study compares these three algorithms for different workloads (with varying data sharing patterns) and system configurations. The key results of the cache consistency study are:

- AACC outperforms both AOCC and ACBL for most of the important workloads and system configurations. AACC achieves this level of performance while maintaining a low abort rate that is competitive with ACBL. This validates the new techniques that are used by AACC.

- AOCC outperforms AACC when the data contention is extremely high. This level of data contention is not found in ODBMS workloads.

- AOCC outperforms ACBL for most workloads and system configurations for write probabilities that are lower than 20%.

- The object level extensions to the AACC cache consistency algorithm allows even object servers to have an efficient and low aborting cache consistency algorithm. This, in turn, allows object servers to compete with page servers.

Chapter 6

Recovery

This chapter first provides a brief background of DBMS recovery by explaining the concepts and terms associated with centralized and client-server recovery mechanisms. It then provides solutions to the following outstanding recovery problems:

- The current client-server recovery solutions cannot handle adaptive data transfers between the server and the clients because they are explicitly designed for page servers. This chapter proposes an adaptive recovery mechanism that allows the server to send either pages or objects to the clients, and that allows the clients to return either pages (pages/log records) or objects (log records that are re-applied by the server to the data pages).

- If an object can be updated at both the clients and the server, then the existing recovery solutions are inadequate [MN94] because they do not have the mechanism to handle this situation. Therefore, in this chapter the client-server version of the ARIES algorithm is extended to support updates at both the clients and the server.

6.1 Recovery Background

The algorithm for Recovery and Isolation Exploiting Semantics (ARIES) [MHL+92] is currently the leading recovery algorithm for centralized DBMS systems. The centralized ARIES recovery algorithm has been adapted for page server client-server architectures [FZT+92, MN94, PBJR96]. This section provides an overview of client-server ARIES algorithm.

6.1.1 Data Structures

The centralized ARIES recovery algorithm uses the following data structures:

- **Log Record:** In general, log records capture the changes to data items as a result of application updates. The log records must be written onto persistent storage before a transaction can successfully commit. During failure or application-initiated rollbacks, log records can be used to redo or undo changes to bring the database back to a stable state. In addition to the data field (before and after difference image), each log record contains a Log Sequence Number (LSN) field, a log type field, a page ID field, a transaction ID field, and a previous log record field (PrevLSN). LSN values are monotonically increasing and they are used to identify a log record on persistent storage. The type of the log record indicates whether it is a normal update log record (generated due to an update action), or a compensation log record (generated during undo), or a special log record (non-transaction related log records). The page ID field refers to the relevant database page whose update is captured by the log record. PrevLSN refers to the previous log record written by the same transaction and is useful for performing backward traversals of the logs during rollback or undo processing.
- **PageLSN:** Each database page contains a PageLSN field. Once an update has been performed, the value of the PageLSN field is set to the LSN of the log record corresponding to the update.

- **Dirty Page Table:** During normal processing, the dirty page table lists all of the updated pages which reside in the buffers but have not been written back to disk. It optimizes the amount of persistent log that needs to be examined during recovery processing. The dirty page table information is written to disk as part of the DBMS initiated checkpointing process. For each data page, it contains the PageID and the RecLSN fields. The RecLSN field contains the LSN of the first log record which has made the corresponding data page (represented by PageID) dirty. The minimum of the RecLSNs of all the pages is known as RedoLSN. RedoLSN is calculated during the beginning of recovery processing.

- **Transaction Table:** The transaction table maintains the list of active transactions. For each transaction, the transaction table contains the state of the transaction (whether it is in-doubt or is unprepared) and it also contains the LastLSN field (LSN value of the latest log record written by the transaction).

- **SaveLSN:** This is the LSN value of the latest log record written during a program initiated savepoint operation. The notion of savepoint is used during application rollback. The application rollback processing starts from the SaveLSN point.

6.1.2 Recovery Processing Modes

The centralized ARIES recovery algorithm uses the following processing modes:

- **Normal Processing:** During normal processing, log records are generated by the server when update actions are performed. Both the dirty page table (if it is the first update on the page) and the transaction tables are updated to reflect the data item update. The PageLSN value on the page is updated and made equal to the LSN value of the corresponding log record. The log records are written to disk before the updated data pages are written back to disk (called write-ahead-logging). All the log records must be written to disk before the transaction can successfully

commit. The Steal/No-Force buffer management policy is generally regarded as the best buffer management policy [MN94], and it is used by most DBMSs. In this policy, the updated pages can be written to their location on disk before the transaction commits (known as stealing), and also the updated pages do not have to be written to their location on disk at the end of the transaction (known as No-Force). During failures, the notion of Steal makes it necessary to undo the updates performed by uncommitted transactions, and the notion of No-Force makes it necessary to redo the updates of committed transactions.

- **Checkpoints and Savepoints:** The DBMS can periodically take checkpoints to reduce the amount of work that has to be done during recovery processing. Checkpoint operations write the information present in the dirty page table and the transaction table to disk as checkpoint records. Subsequently, following a failure, the recovery operation starts from the last successfully written checkpoint record, reducing the amount of persistent log that has to be examined in order to recover. The savepoint operation limits the amount of log that has to be examined during a transaction rollback operation. Checkpoint operations are initiated by the DBMS software whereas savepoint operations are usually initiated by user transaction operations. When a savepoint is established, the LSN value of the latest log record written by the transaction is stored in memory as SaveLSN.

- **Application Rollback Processing:** During transaction rollback processing, the DBMS undoes the affects of the log records with LSN values that are larger than the SaveLSN value. Application rollback processing is performed when the transactions abort due to lock conflicts or are explicitly initiated by the application. The rollback operation starts from the LastLSN log record corresponding to the transaction from the transaction table, and the PrevLSN values present in each log record are used to go through all of the log records written by this transaction. For each log record that has been undone, a compensation log record con-

taining redo information is written to the log in order to recover from failures that occur during the rollback operation. The DBMS also obtains a latch on the data page when the undo operation is performed on the page. This latch is released once the undo operation on the page has been completed.

- **Failure Restart:** ARIES makes the following three passes over the log during the failure recovery process:

  1. **Analysis Pass:** The DBMS starts from the last successfully written checkpoint record. A forward pass is made from the oldest to the latest log record, updating the dirty page table and the transaction table to capture the affects of updates that have occurred since the last checkpoint operation. The DBMS also establishes the starting point for the redo pass. The minimum of the RecLSNs in the dirty page table is determined as the RedoLSN and this is the starting point for the subsequent redo pass.

  2. **Redo Pass:** This pass starts from the RedoLSN. A forward traversal is made of the log records from the oldest to the latest log record. The affect of a log record is redone on a page only if its LSN value is greater than the PageLSN of the corresponding page, and the page has an entry in the dirty page table. In ARIES, the redo operation is also performed for those transactions that could not successfully commit before the failure occurred. This, in turn, makes the undo pass an unconditional one.

  3. **Undo Pass:** During the undo pass, the DBMS undoes the affects of all those active transactions which had yet to commit at the failure point. Similar to the rollback operation, compensation log records are generated during the undo operation. The behavior of the undo pass is similar to the transaction rollback.

6.1.3 Client-Server Recovery Extensions to ARIES

As explained in Chapter 2, this dissertation only considers recovery mechanisms where the log records are stored persistently at the servers. Log records are not stored on local client disks because this makes the server dependent on clients during its failure restart processing, which is not acceptable for most client-server installations. The clients can return both the updated pages and the log records (ARIES-ESM, ARIES-CSA) [FZT+92, MN94], or just the updated pages (whole-page logging) [WD95] or just the log records (redo-at-server) [WD95]. In centralized systems, the server generates unique and monotonically increasing LSN values for the log records. However, in a client-server system, it is too expensive to generate LSNs at the server and to ship them to the clients. Therefore, the page server systems let each client generate monotonically increasing LSNs. In centralized DBMSs, the LSNs also represent the address of a log record within the log. However, in a client-server system, separate log address fields are used to complement LSN fields and these quickly identify the location of a record within the log.

To offload work from the server during transaction rollback processing, it is desirable for the clients to participate in transaction rollback processing [MN94]. Moreover, in the client-server version of ARIES, the clients have the option of returning updated pages to the server at commit time (Force option), or the clients can return the updated pages only when the client buffer is full (No-Force option). However, the server uses the Steal/No-Force buffer management policy. The client ensures that updated pages are never sent to the server without their corresponding log records. In ODBMSs, since the same object can be updated multiple times within a transaction, log records are generated either at commit time, or when data are flushed from the client buffers [WD95]. Before an object is updated for the first time, the client stores the pre-updated (original) copy of an object in its log buffer. Then either at transaction commit time, or when the log or data buffers are full and logs have to be flushed, the client performs a *difference* operation between the pre-updated copy of the object and the current version of the object, and it

generates a log record that contains both undo and redo components.

In addition to server checkpoints, the clients can also take checkpoints to speed up restart processing after client failures. However, to ensure correctness, a server checkpoint also results in the clients taking a checkpoint (co-ordinated checkpoint) [MN94]. In the client-server environment both the server and the clients maintain a dirty page table and a transaction table. Finally, both the server and client failure recovery operations use the standard ARIES 3-pass approach which is performed at the server.

6.2 Hybrid Server Recovery Solution

In the hybrid server architecture proposed in this dissertation, both the server and the clients can transfer pages and/or objects among themselves. The existing recovery solutions cannot handle such adaptive data transfer behavior. Thus a new recovery protocol is proposed, which can also be used by object servers. Currently, there does not exist a published recovery solution for object servers that employs the efficient Steal/No-Force buffer management algorithm. This section describes the new problems and their solutions.

- **Absence of pages at the client:** The log records generated at the client, the client dirty page table, and the state of a page with respect to the log (PageLSN) all require page-level information. Each generated log record contains a log sequence number (LSN). The LSNs are generated and handled in the same manner as in ARIES-CSA. Each page contains a PageLSN, which indicates whether the impact of a log record has been captured on the page. In hybrid servers, objects can exist at the clients without their corresponding pages. Hence, the page-level information might not always be available at the clients. The hybrid server passes to the client the PageLSN and the page id information along with the requested data. After the client receives a group of objects, in addition to creating resident object table (ROT) entries, it also creates the resident page table (RPT) entry. For each received object, the client stores the

PageLSN in the corresponding page entry in the RPT. This allows the client to generate LSNs for the log records corresponding to the page, and also RecLSN values for the page in the dirty page table. RecLSN refers to the log record of the earliest update on the page that is not present on disk. Thus, even though the clients might have only objects and not their corresponding pages in their caches, the clients still keep track of the necessary recovery information for the objects at page-level.

- **Presence of updated objects at the server:** The updated objects returned by the clients are stored in the server MOB and they are installed on their corresponding home pages in a lazy manner using a background thread [Ghe95] (the details of MOB flushing is described in Chapter 3). The pages corresponding to the updated objects might not be residing in the server page buffer. Therefore, it is necessary to keep track of the state of the updated objects in the MOB with respect to the log records. That is, if a client fails and the server is doing restart processing, then the server needs to know the state of the objects in the MOB in order to correctly perform the redo operations. In page servers, the dirty page table at the server keeps track of the pages in the server buffer. Consequently, in addition to the dirty page table, the server maintains a *dirty object table* (DOT) to keep track of dirty objects. Each DOT entry contains the LSN of both the earliest and the latest log records that correspond to an update on the corresponding object.

- **Fine-Granularity Locking:** In client-server DBMSs, different objects belonging to a page can be simultaneously updated at different client sites. In centralized systems the LSNs are generated centrally, so the combination of PageLSN and the LSN of the log record is sufficient to assess whether the page contains the update represented by a log record. In client-server systems, since the clients generate the log record LSNs, two clients can generate the same LSN for log records pertaining to a page. Therefore, the PageLSN alone cannot correctly indicate whether the page contains the update represented by a particular log record. Two

of the previous page server recovery solutions do not allow the simultaneous update of a page at multiple client sites [FZT+92, MN94]. A more recent proposal [PBJR96] permits this and requires the server to write a replacement log record to the log disk before an updated page is written to data disk. For every client that has performed an update since the last time the page was written to disk, the *replacement* log record contains details (client ID and client specific PageLSN) about the client's update to the page. Thus it overcomes the problems encountered due to the generation of the same PageLSN value at multiple clients. However, the proposed fine-granularity locking solution [PBJR96] does not handle the variable object size case where the object size can dynamically increase. If the size of two objects on the same page is simultaneously increased at two different clients, then the space left on a particular page may not be enough to hold both of the objects. The hybrid server recovery solution also uses the notion of *replacement* log records to allow simultaneous updates to a page at multiple client locations. In addition the following steps are necessary to allow for simultaneous updates to variable sized objects:

- At the server, if the space on a page is not enough to hold the updated object, the server moves the object to another page. The server updates the LOID-to-POID mapping data structures and writes a *hasBeenMoved* log record to the log disk to keep track of the object re-location.
- When sending the object updates to the server, the clients send the LOID information of the object that has been updated, which is used to determine the new location of the updated object. The LOID-to-POID mapping information at the client is changed at commit time.
- During the analysis phase of the recovery operation, the server constructs a list of the *hasBeenMoved* log records. This list is used during the redo phase of recovery.

- During the redo phase, if applying a log record to a page can lead to an overflow of the page, then the object is moved to another page. Before generating a *hasBeenMoved* log record, the server first checks to see whether there already exists a previously generated record in the *hasBeenMoved* list. If an entry exists, then the server uses the information about the new page, that is present in the entry, to redo the operation. If a new *hasBeenMoved* entry is being generated, then the LOID-to-POID mapping data structure is updated accordingly.

- **Returning pages or logs to the server:** In the hybrid server architecture, clients return either pages and log records or only log records (redo-at-server recovery). In the latter, the log records have to be installed on their corresponding home pages (ARIES-CSA avoids this). Therefore, each log record is classified at the client as a redo-at-server (RDS) log record or a non-redo-at-server (NRDS) log record. At the server, the RDS log record is stored both in the server log buffer and also in the MOB whereas, the NRDS log record is only stored in the server log buffer. The RDS is stored in the MOB to reduce the installation read overhead. If the client decides to return a page to the server, then it generates a NRDS log record, otherwise it generates a RDS log record. When the client dynamically decides to switch from the redo-at-server mode to ARIES-CSA mode, the following processing is performed at the client and the server:

  - **Changing from Redo-at-Server mode to ARIES-CSA mode at the Client:**

    * **Processing at the Client:** The client ensures that for the subsequent updates, it only generates NRDS log records.
    * **Processing at the Server:** An RDS log record corresponding to the updated page might already be present in the server MOB. Therefore, following the state change from redo-at-server

mode to ARIES-CSA mode at the client, the server can also receive the updated page from either the same client or a different client. Upon receiving the updated page, the server installs the RDS log present in the MOB on the page only if the particular object has not been write-locked by a different client (that is, its corresponding page has also not been write-locked by a different client). If the server receives the corresponding updated page and the page has been write-locked either by the same client or a different client, then the server discards the RDS present in the MOB because the effect of the RDS is already present on the page.

- **Changing from ARIES-CSA mode to Redo-at-Server mode at the Client:**
  - **Processing at the Client:** The client has to ensure that the pages corresponding to the NRDS logs are sent to the server either when the page is flushed from the client data buffer, or if the RDS log is getting sent to the server, or at commit time.
  - **Processing at the Server:** If the page corresponding to the RDS log is already present in the MOB, then the server eagerly installs the RDS to the page.

6.3 Updates Performed at both Clients and Server

Existing client-server recovery solutions cannot handle the case when updates are performed at both clients and the server. This section first discusses the new recovery issues that are encountered when updates are performed at both the clients and the server and then proposes solutions to these issues within the context of an ARIES-style client-server recovery algorithm.

- **Correct Order of Log Execution:** Since the same data item can be updated at both the server and the clients within the same transaction, it is necessary to ensure the correct order of log application during rollbacks and failure recovery. For example, if a data item was updated first at the client, and subsequently at the server, then during rollback processing it is necessary to ensure that the effects of the server updates are undone before the effects of the client updates are undone. In order to solve this issue, it is necessary to have some co-ordination between the server and the client LSN generation process.

  During the application program execution, if the server passes control to the client, or the client passes control to the server, they also pass the LSN values of the latest log records generated for the updated pages to each other. This, in turn, helps the receiver of the LSN value to ensure that the subsequent log records have an LSN value that is greater than the current LSN value. In order to ensure that the address of the previous log record (used during rollback processing) is properly set, when a client transfers application execution control to the server, it also returns the log records that it has generated along with the updated data to the server. This allows the server to properly set the address of the previous log records.

- **Client Participation in Rollback Processing:** It is desirable for the clients to be also responsible for rollback processing because this offloads work from the server. The notion of *transfer-control* log record helps to facilitate client participation in rollback processing in a hybrid function-shipping/data-shipping environment. When a client passes control of application execution to the server, it passes along the log records it has generated along with the updated data. The server then generates a special log record known as the *control-transfer* log record. The control-transfer log record is positioned in the PrevLSN log chain for the particular transaction. The control-transfer log record's type field is set to *control-transfer*. The PrevLSN field of this log record is set to the

LSN of the last log record generated for the transaction at the client. If the server transfers control to a client, the server generates another control-transfer log record and the control-transfer record becomes the previous log record to the logs that will be generated at the client. When the server encounters a control-transfer log record during rollback processing, it passes on control to the client along with the appropriate log records (up to the previous control-transfer log record, or the saveLSN log record) and the updated data. Similarly, when the client encounters a control-transfer log record during rollback processing, it passes on control and the updated data to the server. Thus, the *control-transfer* log record transfers the control of rollback processing between the client and the server.

Apart from providing a recovery solution that supports adaptive client to server data transfer mechanism, this chapter has provided recovery solutions to two outstanding recovery problems that can be used by existing ODBMSs. The hybrid server recovery solution proposed herein can be used by the existing object servers, and the recovery solution to handle updates at both clients and the server can be used as part of the emerging hybrid function-shipping/data-shipping architectures. The performance of the adaptive (redo-at-server and ARIES) recovery solution proposed in this chapter is compared with both the ARIES and the redo-at-server recovery solutions in Chapter 8 of this dissertation.

Chapter 7

Experimental Setup

This chapter presents the experiment organization that is used to measure and compare the performance of the different algorithms and architectures. The algorithmic details of the different system components have already been presented in previous chapters. Simulator and workload setups are the two key components described in this chapter. The experimental environment used in this dissertation builds upon the setups used by previous client-server performance studies (e.g. [CFLS91, FC94, CFZ94, ZC98, AGLM95, Gru97, WD94, WD95, KK94, Ghe95, CDN93, DFMV90]).

7.1 System Setup

A simulator was built using the SMURPH [Gbu96] simulator package to measure the performance of the different client-server algorithms and architectures. The simulator consists of client processes, a server process, a network process and separate disk processes for each of the disks (Figure 7.1). Each of these processes run concurrently during the simulation. The workload generator is a separate process that produces the input work traces that are read by the clients. The server process gets its input strictly from the clients via the network. It produces a stream of <*object id, object size, read/write*> tuples that emulate the data access pattern of applications running at a client. The workload generator uses a random number generator to determine the pages

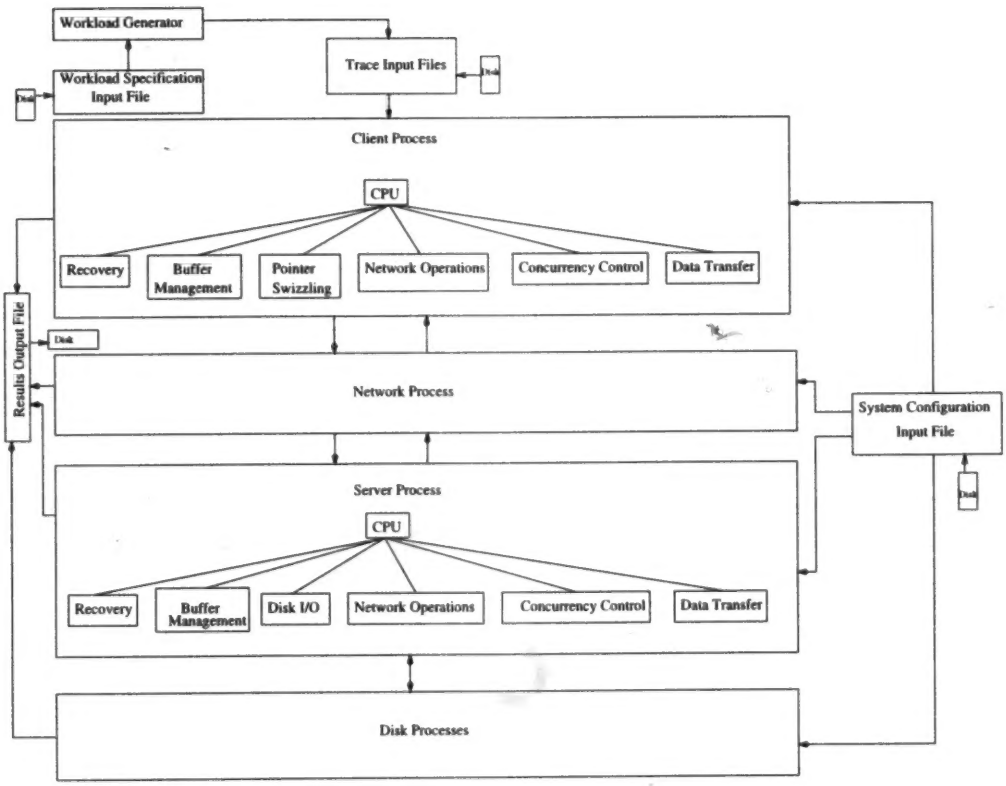


Figure 7.1: Simulator Setup

and the objects within the database, when it generates the input work tuples.

Time is spent in the simulation by the client applications in the following different ways:

- Processes wait in the disk, CPU and network queues.
- Time is spent when processes perform work (application processing, message processing, locking/recovery/buffer management operations) at the client or the server. Time is also spent when disks perform I/O and messages propagate over the network. The work performed at the client and the server is measured as CPU time and it is calculated by multiplying CPU instruction path length for the task by the CPU speed (in millions of instructions per second (MIPS)). The CPU instruction path lengths were calculated by running test programs (testing specific operations such as sending a network message or measuring a page fault etc), on workstations with known SPEC ratings. The disk time is calculated by measuring the seek, rotational delay, and transmission times to transfer a block of data from the disk. The network time is calculated by multiplying the amount of data transferred by the network bandwidth in mega bits-per-second (Mbps).
- Client processes can remain blocked due to pending data, lock, latch and commit requests at the server, or due to conflicting locks held by other clients.

The simulator maintains a global time clock and a global work queue. Each of the new work items generated by the client, server, network and disk processes is put into the global work queue. The global time clock is incremented as the items are removed from the global work queue. Since work can be performed in parallel, by the server, disks, network and the client processes, the global time clock is only incremented when new work (whose end time is not less than or equal to a previously scheduled work item) is performed.

The system parameters and their default values are listed in Figure 7.2. Most of these tasks and their associated costs are the same as those used in

Parameter	Description	Value
Client CPU Speed	Instr rate of client CPU	50 MIPS
Server CPU Speed	Instr rate of server CPU	100 MIPS
ClientBuffSize	Client buffer Size	1 to 12% DB Size
ClientLogBuffSize	Client Log buffer	1 to 2.5% DB Size
ServerBuffSize	Server Buffer Size	1 to 50% DB Size
ServerDisks	Disks at server	4 disks
FetchDiskTime	General disk access time	3322microsecs/Kbyte
InstDiskAccessTime	MOB disk I/O time	1288microsecs/Kbyte
FixNetworkCost	Fixed number of instr. per msg	2000 to 10000 cycles
VariableNetwork Cost	Instr. per msg byte	2 to 7 cycles/byte
Network Bandwidth	Network Bandwidth	10 to 155Mbps
DiskSetupCost	CPU cost for performing disk I/O	5000 cycles
CacheLookup/Locking	Lookup time for objects/page	300 cycles
Register/Unregister	Instr. to register/unregister a copy	300 cycles
Hardware Swizzling	Pointer Swizzling Cost Per Page	50000 cycles
DeadlockDetection	Deadlock detection cost	300 cycles
CopyMergeInstr	Instr. to merge two copies of a page	300cycles/object
Software Swizzling	Swizzling Cost Per Pointer	80 cycles
Database Size	Size of the Database	2400 pages
PageSize	Size of a page	4K to 8K
Object Size	Size of an object	100 bytes to 1 Kbyte
GroupFormCost	Group FormingCost per Object	100 cycles
NumberClients	Client Workstations	12
Indirection Cost	Ptr indirection Cost per Access	15 cycles
Delay Probability	Probability for delaying Message	50 %
Delay Time	Time a message is delayed	0 msec
Software Unswizzling	Unswizzling Cost Per Pointer	30 cycles
LogProcCost	Logging Data Structures Update	50 cycles/Log

Figure 7.2: System Parameters

the previous client-server performance studies [CFLS91, FC94, CFZ94, FCL97, AGLM95, Gru97]. For new costs that have been introduced in this dissertation, a description of how these costs were calculated is provided in this chapter. For well-established costs which were used by the previous performance studies, appropriate references are provided along with a brief description of the cost.

7.1.1 Client Process

The client process performs concurrency control, recovery, buffer management and pointer swizzling operations. It is also responsible for sending and receiving messages via the network. The input work comes to the clients as a stream of object identifiers. The client sends data and lock requests to the server via the network. Client processes do not manage any disk processes. The *NumberClients* parameter determines the number of clients and it is set to 12 to ensure that the network, disk and CPU resources are not saturated [CFZ94, FC94, AGLM95, Gru97], ensuring that the performance characteristics of the different algorithms are not masked. The client CPU contains high priority and low priority input queues. The high priority input queue is used to manage system requests such as message processing, lock processing and recovery processing. The lower priority queue is used to manage the user application program. The high priority CPU queue is managed using a first-in/first-out policy and the low priority queue is managed using processor sharing [CFZ94].

The different CPU costs that are used by the client process are:

- *CacheLookUp* cost [CFLS91, CFZ94, AGLM95, FCL97] is the overhead to lookup pages or objects in the client data structures (ROT/RPT).

- *Register/Unregister* cost [CFLS91, CFZ94, AGLM95, FCL97] is the overhead of bringing in and removing pages/objects from the client cache. During the client cache loading and unloading the ROT and RPT data structures are modified. When an object group is brought into the client

cache, this is the cost of disassembling each object from the object group (that is loading the object into the client cache and registering the object in the resident object table).

- *Lock/UnLock* cost [CFLS91, CFZ94, AGLM95, FCL97] is the overhead for locking and unlocking pages/objects at the client.

- *CopyMergeInstr* [CFZ94, Gru97] is the overhead of copying an object from its page in the page buffer into the object buffer (at the client).

- The client process manages data buffers, log buffers and pointer swizzling buffers. The size of these buffers is stated relative to the size of the working set. It has previously been determined [CFLS91, CFZ94, AGLM95] that the relative size of the buffers with respect to the client working set size is more important than the absolute size of these buffers. This is because the relative size determines the performance characteristics of data transfer, buffer management, pointer swizzling, and cache consistency algorithms. The data buffer acts as a client cache.

- *BuffCoalesceCost* is the overhead associated with coalescing the objects in an object buffer when dealing with variably-sized objects. The coalescing cost consists of combining the buffer frames into larger units.

- The *log buffer* is used for storing the log records. *LogSize* is the size of each log record generated at the client. The size of the log record varies depending upon the size of the update and the size of the object. For a 100 byte object, the size of the log record (including the log record overhead) is 75 percent of the object size. The log record size is similar to the size used by the previous recovery studies [FZT+92, WD95]. The *LogProcCost* is the cost associated with updating the logging/recovery data structures at the client.

- The software pointer swizzling mechanism incurs the *PtrIndirectionCost*, which is the overhead associated with finding the pointer to the target object in the indirection table. It also incurs the *PtrSwizzle* and

PtrUnSwizzle costs. The *PtrSwizzle* cost is the overhead to replace the object identifier with the memory pointer. The *PtrUnSwizzle* cost is the overhead to replace the pointer with the object identifier.

- *PtrAccessCostPerPage* is the overhead incurred by the hardware swizzling mechanism when a page is brought into the client buffer [WD95]. This overhead includes the page faulting cost, pointer swizzling cost, CPU overhead for managing hardware swizzling data structures, and the mmap operation for setting page access control protections. The *PtrAccessCostPerPage* value used in this dissertation is similar to the value used in a previous pointer swizzling study [WD94].

- The client encounters *VariableNetCost* and *FixedNetCost* messaging overhead for sending and receiving every message [CFLS91, CFZ94, FC94, AGLM95]. *FixedNetCost* is independent of the message size, whereas the *VariableNetCost* is a per byte overhead associated with sending or receiving a message. These two costs have been re-examined in this dissertation to assess the impact of newer hardware and software on the costs.

7.1.2 Server Process

The server process performs cache consistency/concurrency control, recovery, buffer management and disk I/O operations. It sends data, lock request grants, and callback messages to the clients via the network. The server also contains high priority and low priority CPU input queues that are managed in the same manner as the client CPU input queues. The input work comes to the server from the clients via the network. The server manages a log staging buffer, a page data buffer, and a modified object/page buffer. The sizes of these buffers are specified in Figure 7.2 as a percentage of the database size. The server process encounters the following CPU overheads:

- *VariableNetCost* and *FixedNetCost* overhead are also incurred by the server for sending and receiving network messages. These overheads are

the same as described in Section 7.1.1.

- *DiskSetupCost* [CFLS91, CFZ94, Gru97] is the CPU overhead that is incurred by the server when the client requested data are not present in its buffer and the server initiates disk I/O.

- *CopyMergeInstr* [CFZ94] overhead is present when an object is installed on to its corresponding home page.

- When the hybrid server or object server form an object group, they incur *GroupFormCost* overhead for each object that includes the server overhead of performing the necessary calculations to partition the page into equally-sized sub-segments of the desired object group size, and to determine the sub-segment that contains the desired object. This cost also includes the cost of creating an object group header that describes the objects in the group, and the overhead of setting up locking (lock group) and recovery (PageLSN) information in the object header for each object in the object group.

- The server also maintains lock information at both the page and the object level. Similar to the client process, there is a *Lock/Unlock* cost [CFLS91, FC94, CFZ94, AGLM95] associated with each lock/unlock operation at the server.

- *Deadlock* detection cost is encountered by the server when it detects locking conflicts. A deadlock detection technique similar to the one used by EOS [BP97] is used.

- *Register/Unregister* cost [CFLS91, CFZ94, AGLM95, FCL97] is the overhead of bringing in and removing pages/objects from the server buffers. During the server cache loading and unloading the ROT and RPT data structures are modified.

- The server also incurs *CacheLookUp* [CFLS91, FC94, CFZ94, AGLM95] overhead when accessing data from the page or MOB buffers.

- *LogProcCost* is the cost incurred at the server for processing an incoming log record from the client. This is the cost associated with updating the logging/recovery data structures at the server.

- *BuffCoalesceCost* is the overhead associated with coalescing buffer frames when dealing with variable sized objects in the MOB. The coalescing cost consists of combining the buffer frames into larger units.

- *LOID to POID mapping cost*: A cost of 300 instructions has been alloted in order to traverse the B-tree data structure and find the POID corresponding to a particular LOID.

7.1.3 Disk Process

Each disk connected to the server has its own corresponding disk process. The disk processes receive their work from the server process. When the server process has to perform an I/O operation, it uniformly selects one of the connected disks. The server manages a separate log disk process that only handles logging operations. Each disk process contains a FIFO input queue. The number of disks was selected to ensure that disk contention does not mask or alter the results of the performance experiments where disk performance is not supposed to be an issue. This dissertation uses a slow disk latency and a fast disk latency. The fast disk latency is used for the installation read/write operations of the data stored in the MOB. Installation I/O operations can be intelligently scheduled by the server [Ghe95]. The slow disk latency is used for the normal read operations that are not scheduled by the server. The disk latency numbers were calculated using the Seagate Barracuda disk drive with an average seek time of 8.75 msec, a rotation time of 8.33msec and an average transfer rate of 0.37msec for 4Kbytes [Gru97]. The slow disk latency used herein is 3.322 msec per kilobyte and was calculated using a random workload on 4 Kbytes pages. The fast disk latency is 1.288 msec per kilobyte. The fast disk latency was calculated by intelligently scheduling a group of installation I/Os. This approach is same as the one used in a previous cache consistency

study [AGLM95]. By flushing a large enough portion of the MOB (10%) at a time, and by intelligently scheduling these I/Os, the installation I/O cost is reduced. A previous study on disk scheduling [SCO90] has shown that if the number of pending I/Os is not too small, then it is possible to intelligently schedule the I/Os to reduce seek and rotational delays.

7.1.4 Network Process

The network overhead consists of the software transmission CPU cost and the on-wire propagation cost. The on-wire propagation cost is related to the network bandwidth (100 Mbps or 10Mbps). The software transmission cost is the overhead incurred by the client or server CPUs for sending or receiving a network message. The network model is that of a switched network in which each client has a point-to-point connection with a network switch, and the network switch, in turn, has a point-to-point connection to the server. Thus, messages can incur network-related delays when transferred between the switch and the server and vice-versa. It is assumed that the messages incur negligible switching delay. The software transmission cost overhead (presented in Figure 7.2) was determined by sending varying sized messages between two workstations over switched Ethernet. The software transmission overhead varies depending upon the network protocol (TCP versus UDP) and the type of operating system (AIX, Solaris, SunOS, IRIX). The technique used to calculate the fixed and variable overhead for sending messages is similar to the techniques used previously [Gru97]. The round-trip latency for a small sized and a large sized messages are measured on an isolated network. These numbers are then halved to get the 1-way latencies, from which the 1-way wire times are subtracted. This gives the CPU cost which is then halved to assign half of the cost to the sending processor and the other half to the receiving processor. Finally, the SPECInt92 or SPECInt95 rating (whichever number is available) of the machine is used to calculate the number of CPU cycles.

Since many of the emerging application domains operate on the Internet, it is important to assess the impact of unpredictable network delays that are

Parameter	Setting
Transaction size	180 to 220 objects
Per Client Hot Region	5% DB Size
Object write probability	0% to 20 %
Read access think cost	50 cycles/byte
Write access think cost	100 cycles/byte
Think time between trans	0

Figure 7.3: Workload Parameters

common on the Internet, especially the impact on cache consistency algorithms. Initial message delay, slow delivery and bursty arrival are the three types of delays examined in a recent WAN performance study [AFT97]. Similarly, network delay is simulated by making the message sending source wait for a specified time before sending the message. The message sending source flips a coin to determine whether a message should be delayed *(delay probability)*. The actual value of the delay *(delay time)* is chosen as an integer multiple of the expected time to send and receive a message.

7.2 Workload

The workload generator used in this performance study is based upon many previously proposed ODBMS benchmarks [CS92, CDN93, CDKN94, DFMV90], vendor surveys [Obj98, Ver98] and performance studies [CFLS91, FC94, CFZ94, AFT97, AGLM95, Gru97]. The previous performance studies did not consider all of the system components such as data transfer, pointer swizzling, recovery, cache consistency/concurrency control, and buffer management in an integrated manner. Therefore, even though the workload generators used in those studies contain some useful components that can be re-used for the studies included in this dissertation, it was necessary to design and implement a more comprehensive workload generator. This section describes the different components of the workload generator, and then discusses how these different components interact with each other. The following is a detailed description of the different components of the workload generator:

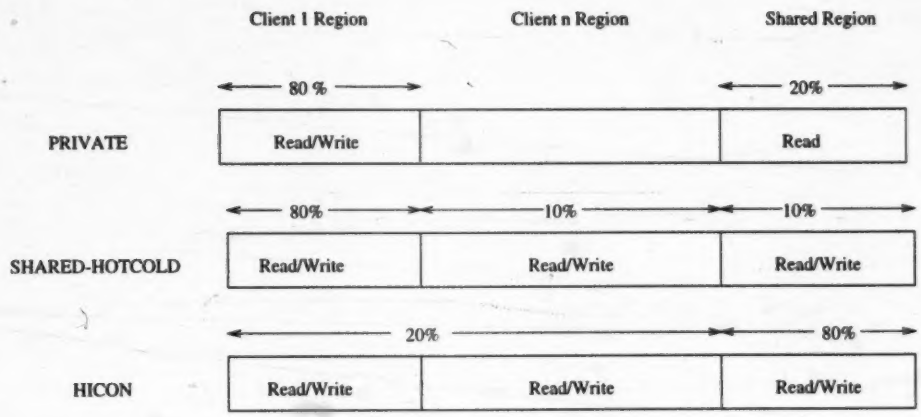


Figure 7.4: Data Sharing Patterns

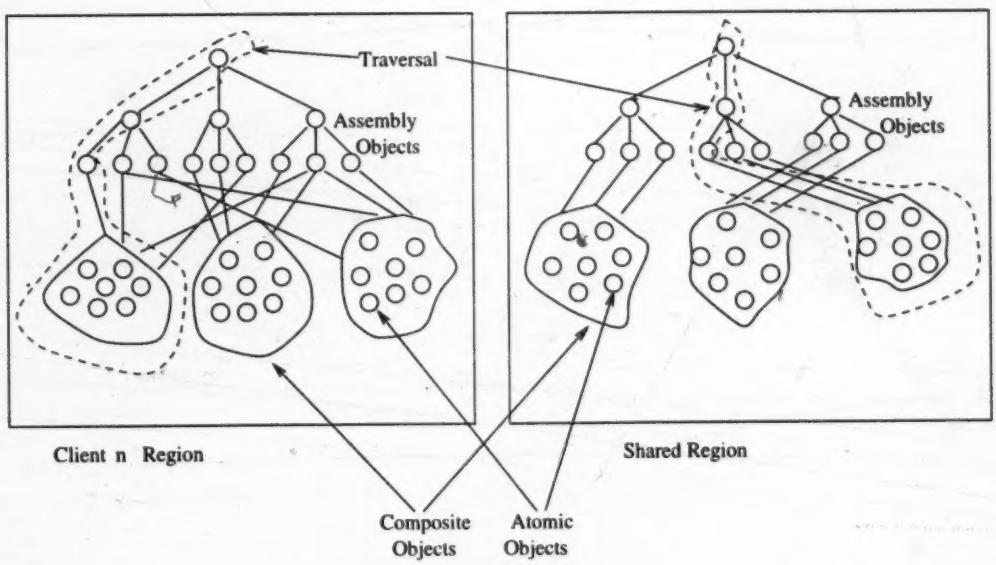


Figure 7.5: Traversals in OO7

- **Traversals:** The workload borrows the notion of traversals from the OO7 [CDN93] benchmark, which is the latest in a series of ODBMS benchmark specifications. It models computer-aided design (CAD) application access patterns. As shown in Figure 7.5, a traversal consists of accessing a sequence of inter-linked objects, each of which are accessed by navigating from the preceding source object in the chain to the target object using the object identifier of the target object that is stored in the source object. In multi-user OO7, a traversal starts at the root node and ends at a composite object, and all of the objects in a composite object are accessed during the traversal. As shown in Figure 7.5, the database consists of client regions and shared regions. Traversal activity in a transaction consists of a set of operations. Each operation consists of object accesses from either a client's region or the shared region of the database. The data sharing pattern (described below) determines the percentage of operations performed on the shared and client regions, respectively. When a region belongs to a client, it means that the client has an affinity towards the data in that region. Client regions can also be called private regions, if clients do not access the regions of other clients. An operation consists of accesses to multiple composite objects and a transaction consists of multiple traversals.

- **Working Set/Database Size:** Each of the client regions in Figure 7.5 is classified as the *hot region* for a particular client. The size of the regions that are simultaneously read and updated by multiple clients contributes towards the data contention level of the system. The size of a client's working set in conjunction with the client buffer size determines whether the client's working set fits into the client buffer. Similarly, the cumulative size of all the client working sets in conjunction with the server buffer size determines whether the combined working sets fit into the server buffer. It is necessary to examine the case when the working sets do not fit into the client and server buffers and also the case when the working sets fit into the buffers. The notion of small, medium and large

databases has been borrowed from the OO7 benchmark specification [CDKN94].

- **Transaction Size:** When multiple clients perform conflicting read and write operations, the size of the transaction also affects the data contention level. The transaction sizes (Figure 7.3) used in this dissertation are similar to the ones used in the previous cache consistency studies [CFZ94, AGLM95] and they also adhere to the general guidelines of the surveys of the ODBMS vendors [Obj98, Ver98].

- **Data Sharing Pattern:** The data sharing pattern dictates the number of read/write and write/write conflicts in the system, and, thus has a major impact on the data contention level of the system. *Shared-HotCold*, *Private* and *HiCon* are the three data sharing patterns examined in this dissertation. These data sharing patterns were developed by the previous client-server ODBMS cache consistency performance studies [CFLS91, CFZ94, FCL97, AGLM95] (see Figure 7.4).

  - **Private :** In the Private data sharing pattern, 80 percent of the traversal operations take place on the client's own private data (hot region) and 20 percent of the traversal operations take place on the shared data. Update operations only occur on the private data. Therefore, there are no read/write or write/write conflicts in this data sharing pattern, which is prominent in the CAD-like workloads [CFLS91].

  - **Shared-HotCold:** In the Shared-HotCold data sharing pattern, 80 percent of the traversal operations take place on the client's own private region, 10 percent take place on the shared region, and 10 percent take place on any other clients' region. Update operations are evenly performed between the client's own hot region, the shared region, and other client hot regions. Therefore, there is an occurrence of read/write and write/write conflicts. Shared-HotCold is a common sharing pattern that occurs in many ODBMS applications

[AGLM95].

- **HiCon:** In the HiCon data sharing pattern, 80 percent of the traversal operations occur on the shared data and 20 percent of the traversal operations occur on the rest of the private data. Update operations are performed in every area. This is a highly skewed data access pattern which is rare in ODBMS workloads [CFZ94]. However, it is useful for testing the robustness of the different algorithms.

- **Clustering Pattern:** The data clustering specification consists of spatial, temporal and access locality values. These parameters are expressed as a value between 0 and 100 percent. A description of these parameters with examples of how they can be combined has been provided in Section 4.1.

- **Page Size:** The workload generator varies the size of the database pages. Most experiments use 4K byte pages which are commonly used by most of the previous ODBMS studies [CFLS91, CFZ94, AGLM95]. However, large 16K byte pages are also used to assess the interaction between large pages and spatial, temporal and access localities.

- **Object Size:** The workload generator varies the size of the objects to assess the impact of fragmentation on object buffer management schemes and also to test the robustness (with respect to grouping accuracy) of the object grouping mechanisms. Object sizes are varied between 100 bytes, 500 bytes and 1 Kbytes. The traversal primarily accesses 100 byte objects. In addition, each traversal also accesses a 500 byte object and a 1 Kbyte object. Previous vendor studies have shown that the object sizes range in the 75 byte to 100 byte range [Obj98]. Since this dissertation does not deal with large objects (greater than the size of the page), object sizes are kept below page sizes.

- **Composite object graph size:** In this dissertation, the composite object graph size (number of atomic objects) is represented by the spatial

locality specification. The spatial locality value is varied between 10 percent and 90 percent of the page size. Thus, for a 4K page size, composite object graphs with sizes ranging from 4 to 32 hundred byte objects are used. ACOB benchmark uses a composite object graph size of 7 atomic objects [DFMV90] and the small database OO7 benchmark [CDN93] uses a composite object graph size of 20 atomic objects.

- **Write probability:** During the traversal operation, as each object is accessed, the write probability is used to determine whether an update operation will be performed on the object. The data sharing pattern determines whether objects in a region are updated. For the private workload, the objects in the shared region are not updated. The write probability values used in the previous cache consistency performance studies [CFLS91, FC94, CFZ94, AGLM95] are also used here, and it varies between 0 and 20 percent.

- **Number of Pointers:** The number of pointers (between the objects) in the database has an impact on the performance of the different pointer swizzling architectures. The number of pointers determines the pointer swizzling and unswizzling overhead incurred by the different approaches. Similar to the OO7 benchmark, this study uses 3 pointers per object.

- **Abort variance:** When a transaction aborts due to locking conflicts, a decision has to be made as to whether the aborted transaction should access the same set of objects as the original transaction, or whether it should access a different set of objects [Gru97, ACL87]. The notion of *abort variance* is used to control how many of the objects accessed by the original transaction are re-accessed by the re-start of the abort transaction. An abort variance of 0 percent means that the same set of objects are accessed during the transaction re-run, and an abort variance of 100 percent means that a completely different set of objects are accessed from the original aborted transaction. An abort variance of 50 percent means that the restarted transaction will use the same objects as

the original transaction 50 percent of the time. That is, from the point of failure (the object access which caused a locking conflict leading to the abort), there is a 50 percent chance that the same set of subsequent objects in the original transaction sequence will be accessed again by the restarted transaction. If the accessed object in the original transaction was in the client's hot (cold) region, the new object is also selected from the client's hot (cold) region. The abort variance is varied between 0 and 100 percent.

Figure 7.6 discusses how the different workload components described above are integrated in the workload generator. Similar to previous performance studies, the workload generator uses a uniformly distributed random number generator when dealing with probabilities that are required by the different components of the workload.

7.2.1 Server Work Allocation

Clients can request the server to process embedded user functions, which can contain traversal operations. These traversal operations can perform both read and write operations at the server and they can use the data sharing patterns described above. The *work allocation* parameter controls the percentage of traversal operations that are performed at the client and at the server.

7.3 Simulator Validation

The simulator has been validated by comparing the results obtained using this simulator with the results obtained from previous performance studies. As shown in Figure 7.7, different functional components of the simulator have been validated against different implementations and simulation studies. The simulator used in this study has been validated by using the costs in the previous studies to ensure that this simulator can duplicate the results obtained in the previous study. The following key results from the previous performance studies have been duplicated using this simulator:

```
For each client do
{
/*Execute the prescribed number of transactions   */
  For each transaction do
  {  Determine the transaction size
     Determine the number of operations
     For each operation do
      {
        Determine the number of objects accesed
        Vary the object access pattern according to
            Spatial locality
            Temporal locality
            Access locality
        Determine whether it will operate on private or shared region

        For each object within the operation do
        {
            Determine the object size
            Determine whether will perform a read or a write operation
        }
      }
  }
}
```

Figure 7.6: Workload Generator Pseudo-Code

System Components	Implemented Systems	Simulations
Data Transfer	[DFMV90] [LAC+96] [ZC97]	[CFZ94]
Cache Consistency	[LAC+96] [ZC97]	[CFLS91] [FC94] [CFZ94] [AGLM95] [Gru97]
Pointer Swizzling	[WD94]	
Recovery	[WD95]	
Client Buffer Management	[KK94]	
Server Buffer Management	[Ghe95]	

Figure 7.7: Simulator Validation

- The page server outperforms object servers that transfer single objects between the server and the clients [DFMV90, CFZ94].
- AOCC cache consistency algorithm outperforms ACBL algorithm for Private, Sh-HotCold and HiCon workloads [AGLM95, Gru97].
- Hardware pointer swizzling outperforms software pointer swizzling when the size of the on-disk OIDs in the software pointer swizzling mechanism is larger than the in-memory pointers in the hardware swizzling approach, and due to the absence of pointer indirection, during object access, in the hardware swizzling approach [WD94].
- A client with a dual buffer management mechanism outperforms a client with a page only buffer during bad clustering [KK94].
- ARIES style client-server recovery algorithm outperforms Redo-At-Server style client-server recovery algorithm when there is no modified object buffer present at the server [WD95].
- Sending modified objects to the server is better than sending modified

pages to the server when the server buffer is not contended and the server has a MOB [Ghe95].

- Sending groups of object from the server to the client is better than sending pages when the temporal and access localities are good and when the spatial locality is poor [LAC+96].

- In most cases, adaptive object/page level locking is better than page-level only locking [ZC98, CFZ94].

While conducting sensitivity analysis on the different costs, it was observed that the disk I/O cost and the network message transmission cost are much larger than the other costs, and they have more impact on the overall performance than the others. Therefore, experiments have been run using a range of values for parameters to assess their impact on the performance of the different algorithms. In this dissertation, as well as in many of the previous performance studies [CFLS91, FC94, CFZ94, AGLM95, Ghe95], disk, CPU and network functionality have been modeled at a higher macroscopic level. CPU processing power is in terms of abstract MIPS (millions of instructions per second). It has been recently stated that even though the CPU clock speed is increasing at a fast rate, the L1 and L2 cache misses prevent the applications from realizing a CPU rating that is greater than 100 MIPS [KPH98]. Therefore, the experiment setup uses CPU ratings that do not exceed this value. Modern disk drives have large on-disk caches that are used to pre-fetch data from entire disk tracks. Furthermore, these disk drives also try to intelligently schedule the outstanding I/O requests from the disk queue. Therefore, the impact of these optimizations is to reduce the overall cost of a disk I/O. Since the simulator does not model the disks with these optimizations, the experiments have been run using a range of disk I/O costs to assess the impact of varying disk costs on the overall results.

Chapter 8

Performance Study

In this chapter the new adaptive hybrid server architecture is compared with other leading data shipping ODBMS architectures. The comparison is based on an integrated performance study that looks at the system components data transfer, cache consistency/concurrency control, recovery, buffer management and pointer swizzling in an integrated manner. This is the first *integrated* study in this field and the results demonstrate the interplay among different system components under different algorithms. Data granularity (*page* versus *object*) has a major impact on the performance of the different data transfer, cache consistency, recovery, buffer management and pointer swizzling algorithms. The current ODBMS client-server architectures are either page-based or object-based. The focus of this performance study is to show that the hybrid client-server architecture, which can dynamically adapt between page and object level granularities, is more robust than either exclusively page-based and object-based architectures.

This chapter also presents a performance study of cache consistency algorithms that goes beyond data granularity concerns. The study compares the performance of AACC, which was described in Chapter 5, with other leading client-server ODBMS cache consistency algorithms for different workloads and system configurations. The focus of the study is to evaluate whether AACC has a better combination of performance and abort rate than ACBL and AOCC.

This chapter first presents the cache consistency study because the client-server architectures that are compared in the integrated performance study all use the same AACC cache consistency algorithm. Presentation of the integrated performance study then follows.

Each of the experiments in this chapter describes the system and workload parameters, followed by the primary and secondary graphs. Primary graphs indicate the overall system throughput (commits/second), which is the main measurement. The secondary graphs and tables provide supporting data to help interpret the primary graphs. To ensure the statistical validity of the results, the 90 percent confidence intervals for system throughput in commits/second were calculated using batched means. The confidence intervals were within a few percent of the mean. Each experiment was run three times using three different random number seeds and each run consisted of twenty thousand transactions. The parameters described in Chapter 7 are used for all the experiments unless explicitly noted.

8.1 Cache Consistency Study

In the cache consistency study, AACC is compared with the AOCC and ACBL cache consistency algorithms. The data transfer, recovery, buffer management and pointer swizzling components have been fixed during this study. Lock granularity related issues are not examined at this stage. The server transfers pages to the clients and the clients return updated objects back to the server. The clients use a page buffer and the server contains both a staging page buffer and a modified object buffer. The redo-at-server recovery and software pointer swizzling are utilized by all of the systems under comparison.

8.1.1 Cache Consistency Study Outline

The performance results reported in this section compare the performance of the ACBL, AOCC and AACC cache consistency algorithms by varying the following parameters:

- **Different Data Sharing Patterns:** Private, Sh-HotCold, and HiCon workloads are used to assess the impact of different levels of read/write and write/write conflicts on the performance of different cache consistency algorithms. The data sharing patterns are the primary means according to which the cache consistency study experiment results have been organized.

- **Write Probability:** This is a key parameter which affects the number of messages issued by the different algorithms, and it also influences the number of read/write and write/write conflicts. The write probability is varied between 0 and 20%, which is usually present in ODBMS applications [CK89, Ghe95, CFZ94].

- **Clustering Pattern:** *Spatial*, *temporal* and *access* locality values together constitute the data clustering pattern. Access and temporal localities do not have an impact on the size of the client working set. Similar to the previous cache consistency studies [AGLM95, CFZ94] the workloads use between 10 and 30% spatial locality. The temporal locality has been varied between 0 and 50%.

- **Buffer Sizes:** The client and the server buffer sizes are varied to assess the impact of different buffer sizes on the performance. The buffer sizes are classified as *Small* and *Large*. *Small* client buffer size refers to the case where a single transaction's state does not fit into the client cache. *Small* server buffer size refers to the case where there is buffer contention at the server even during steady state operation. That is, one encounters misses at the server buffer due to the sharing of the server buffer by multiple clients. The size of the server buffer has an impact on the disk utilization of the system. Small client buffers can exist if the client cache is shared between multiple client processes, or if the transactions are very long or if the objects accessed by the transaction are large (multimedia, audio or image). Small server buffer conditions can exist when the combined working sets of the clients is greater than the server buffer

size. The previous performance studies comparing AOCC and ACBL [AGLM95, Gru97] only focused on large client caches, where a client's entire transaction state (data and logs) fit into the client cache. This is favorable to an optimistic algorithm because, during abort processing almost all of the relevant objects already reside in the client cache, making abort processing inexpensive.

- **Network Delay and Network Speed** The network delay due to initial message delay, slow delivery and bursty arrival are varied to assess the impact of delays on the performance of the three algorithms [AFT97]. In reality, the *delay probability* and *delay time* values can vary significantly depending on the network traffic, geographic location, and intermediate node down times. Previous cache consistency studies have not assessed the impact of network delay on performance. The network speed has also been varied to assess the impact of network bandwidth and software message processing overheads on the performance of algorithms that send explicit lock escalation messages. A range of bandwidth and software message processing overheads are used corresponding to slow, fast and normal network speeds. The slow speed corresponds to 10Mbps network, 10000 cycles/message fixed CPU cost and 7 cycles/byte message variable CPU cost. The normal speed corresponds to 100 Mbps network, 6000 cycles/message fixed CPU cost and 4 cycles/byte message variable CPU cost. The fast speed corresponds to 155 Mbps network, 2000 cycles/message fixed CPU cost and 2 cycles/byte message variable CPU cost.

- **Abort Variance:** Most of the experiments have been run with an abort variance of 50% because an abort variance of 100% favors ACBL and an abort variance of 0% favors AOCC.

- **CPU Speed:** The client and the server CPU speeds have been varied in the experiments. Experiments have been run using slow CPU and fast CPU speeds. Slow CPU speeds are similar to the CPU speeds used in

previous cache consistency studies [AGLM95, CFZ94], and the fast CPU speeds are current CPU speeds.

8.1.2 Private Workload Experiments

In the private workload, the clients only perform updates on their private hot regions and do not perform any updates on the shared or other client regions. Private workload is indicative of computer-aided design (CAD) environments where the users perform updates on their private data, but also do reads on shared data. Due to the absence of data contention, no aborts occur in this workload. The write probability is varied on the x-axis for private workload experiments and overall system throughput in commits-per-second is measured on the y-axis. The client and server data buffers are large. The access and temporal localities have been set to 50%. 100% percent of the work is performed at the client and the network speed is set at 100 Mbps. The CPU speeds of the server and the client are 100 MIPS and 50 MIPS, respectively.

Experiment 1: Low Spatial Locality

The objective of this experiment is to assess the performance of the algorithms without any data contention. In this experiment the spatial locality has been set to 20%. As shown in Figure 8.1, AOCC and AACC perform identically and they both outperform ACBL for all write probabilities. In ACBL, the clients send explicit lock escalation messages to the server to obtain exclusive page level locks for every page that is updated and they block until the server responds. In AOCC, all the write notifications are deferred until commit time, while in AACC, the *shared-private* optimization ensures that all update notifications are sent to the server in a piggy-backed manner. As evident in Figure 8.2, ACBL sends more messages than AOCC and AACC because in every transaction ACBL sends a lock escalation message for every page (objects on the page) it updates. Therefore, the higher message transmission and message blocking overhead increases ACBL's write lock acquisition overhead which, in turn, makes its performance lower than AOCC and AACC.

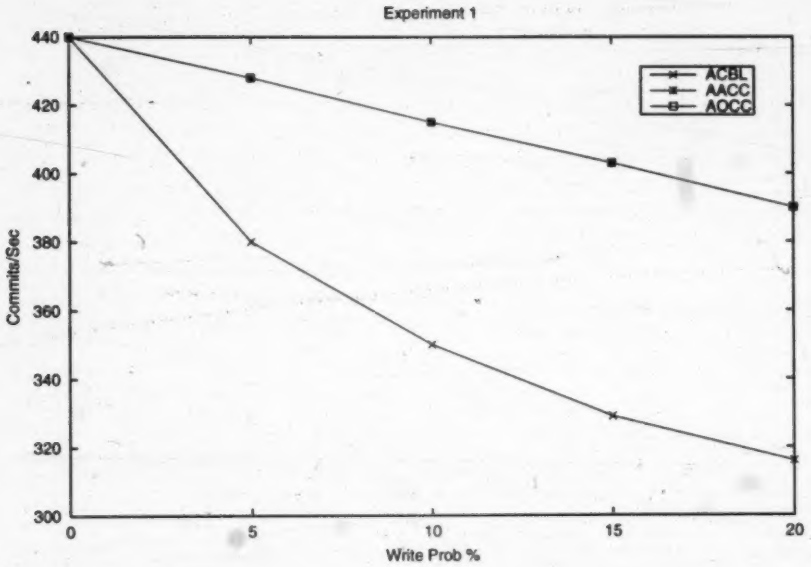


Figure 8.1: Private Workload: Low Spatial Locality

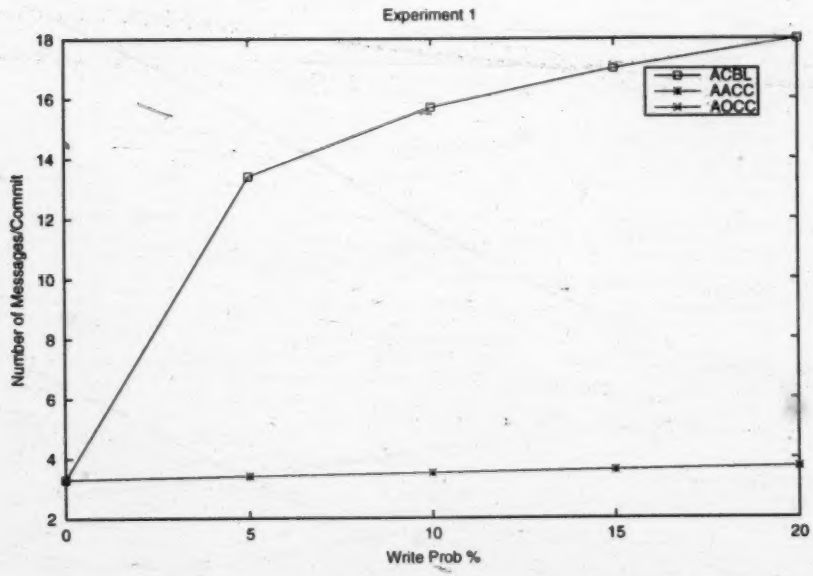


Figure 8.2: Private Workload: Low Spatial Locality

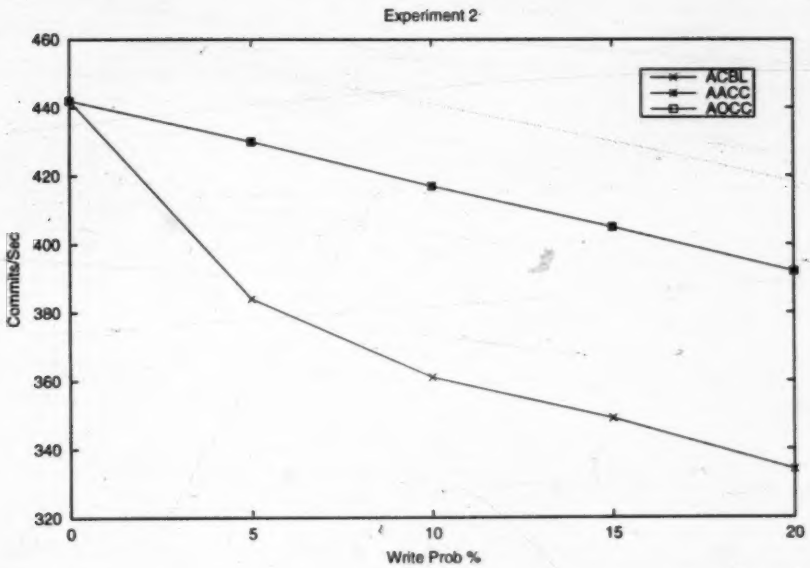


Figure 8.3: Private Workload: High Spatial Locality

Experiment 2: High Spatial Locality

The objective of this experiment is to see whether an improvement in the spatial locality helps ACBL's performance to catch up with the performance of AOCC and AACC because there are fewer pages in the working set of a client. The experiment setup is the same as experiment 1, except that the spatial locality of the client access pattern is set to 90%. As shown in Figure 8.3, the performance of ACBL improves because it sends fewer number of lock escalation messages to the server per every transaction as there are a fewer number of pages in its working set. However, AACC and AOCC still outperform ACBL. Thus, higher spatial locality improves the performance of synchronous cache consistency schemes, such as ACBL, that issue explicit lock escalation messages on private data.

8.1.3 Shared-HotCold Workload

Sh-Hotcold workload data contention level is indicative of the data contention level present in most client caching applications. Therefore, its results are

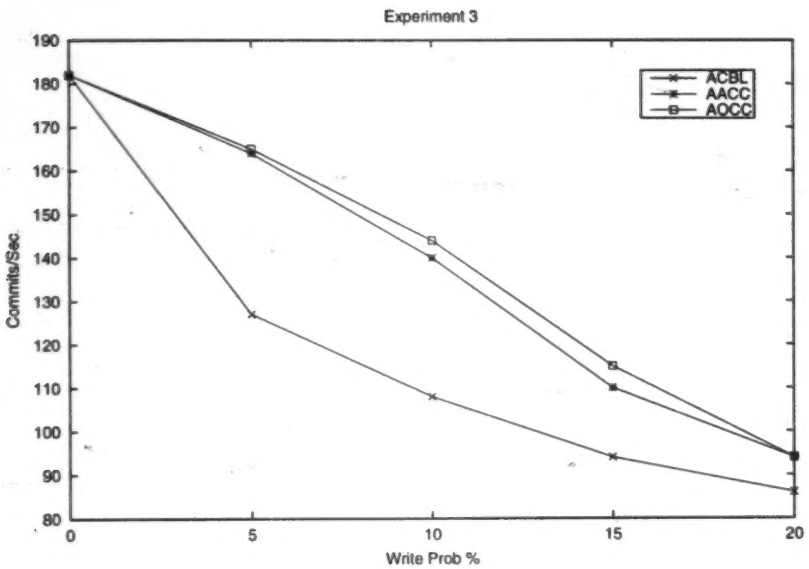


Figure 8.4: Slow CPU: Sh-HotCold

very important. Due to the presence of data contention, stale cache aborts are possible in AOCC, and deadlock aborts are possible in AACC and ACBL. In these experiments, the client and server data buffers are large. The spatial and temporal localities have been set to 50%. The network speed is 100 Mbps.

Experiment 3: Slow CPU

The objective of this experiment is to assess the performance of the different algorithms for Sh-HotCold workload when using slow CPU speeds. In this experiment, the server and client CPU speeds are 50 MIPS and 25 MIPS respectively. These CPU speeds, which are classified as *slow* in this dissertation, were used by the previous cache consistency performance studies [CFZ94, AGLM95], and they are used here to help compare the change in the performance when moving from slow CPU speeds to fast CPU speeds. The abort variance value has been set to 50%. As evident from Figure 8.4, AOCC and AACC outperform ACBL for all non-zero write probabilities. At 0% write probability, since no updates are performed, the performance of all three al-

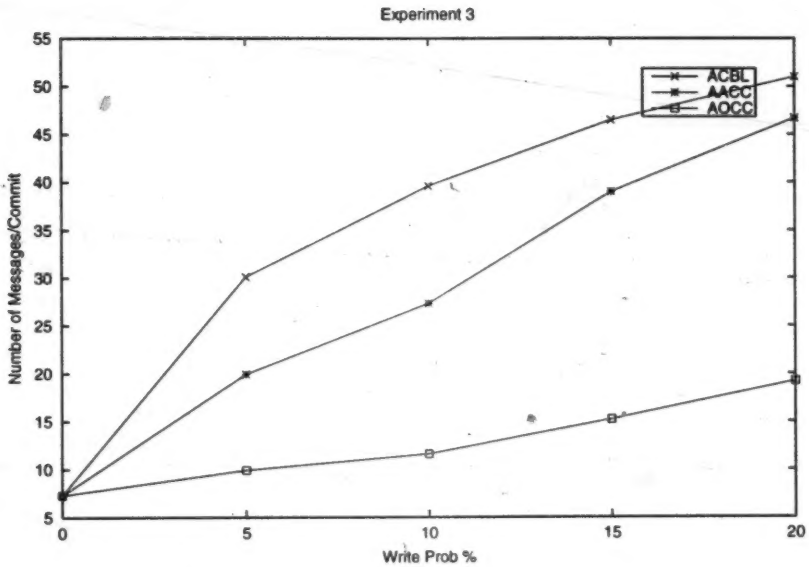


Figure 8.5: Message Count

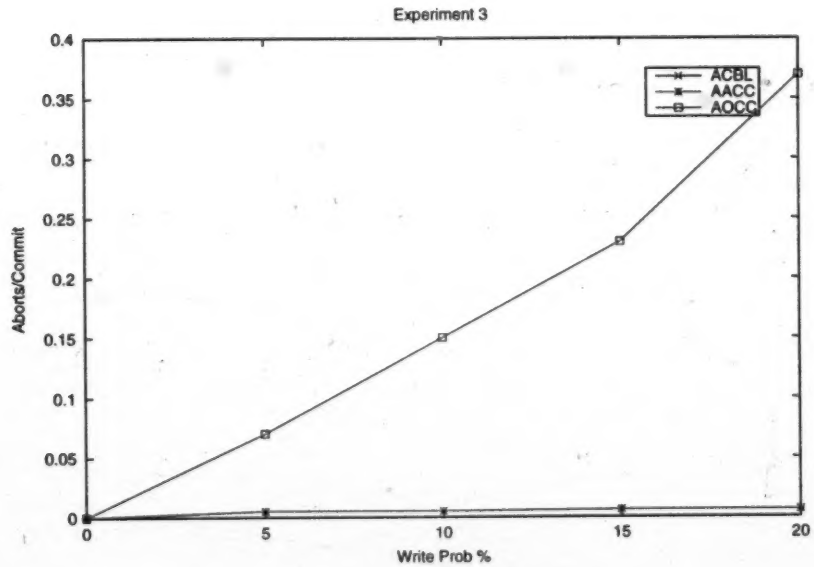


Figure 8.6: Abort Rate

gorithms is identical. Figure 8.5 shows that AOCC sends fewer messages than AACC, and AACC sends fewer messages than ACBL. Thus, the lower message overhead helps AOCC. However, Figure 8.6 shows that AOCC has a higher abort rate than AACC and ACBL. The abort processing overhead associated with this high abort rate degrades AOCC's performance and allows AACC to almost match AACC performance. ACBL is not able to match AOCC performance because ACBL uses synchronous lock escalation messages and, therefore, incurs higher message blocking overhead. Even though AOCC has a higher abort rate, as shown in Figure 8.6, its performance does not degrade drastically because AOCC uses a fast abort processing mechanism that keeps the undo log records in the client cache, and, thus eliminates the need to fetch them from the server. However, as the write probability approaches 18% the number of aborts in AOCC start to become a factor. Here AACC performance is identical to AOCC performance.

AACC outperforms ACBL for the entire range of write probabilities. AACC uses fewer messages than ACBL because, in AACC, the write lock messages for private pages are piggybacked on other messages. Furthermore, in AACC, clients also piggyback callback responses if there are no lock conflicts. Since AACC uses asynchronous lock escalation messages on shared pages, AACC has lower message blocking costs than ACBL. Finally, the deadlock avoidance techniques used by AACC allow it to have an abort rate (as seen in Figure 8.6) which is as low as ACBL's abort rate.

Experiment 4: Fast CPU

The objective of this experiment is to assess the performance of the different algorithms for the current fast CPU speeds. As shown in Figure 8.7, AACC outperforms AOCC and ACBL. This experiment's setup is the same as Experiment 3, except the server CPU speed is 100 MIPS and the client CPU speed is 50 MIPS. In this dissertation, these are considered as fast CPU speeds. This result is important because a previous study comparing AOCC and ACBL [AGLM95] indicates that AOCC always outperforms avoidance-based cache

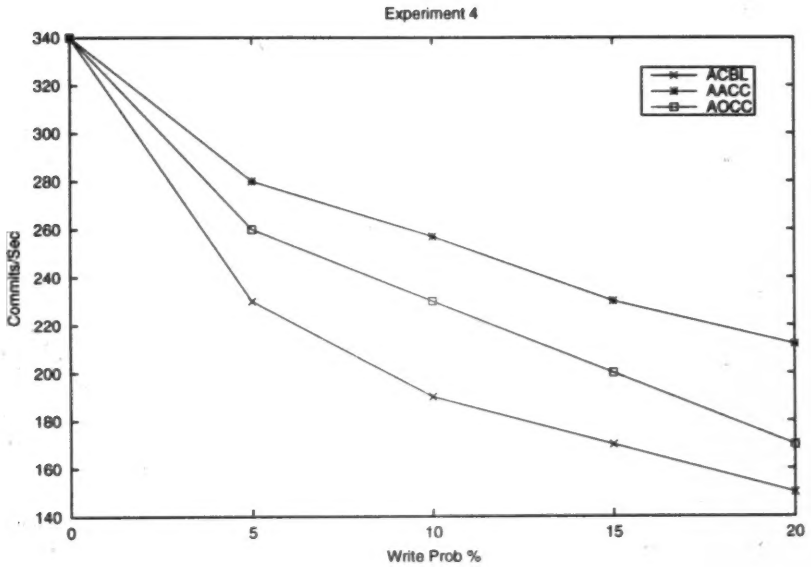


Figure 8.7: Fast CPU: Sh-HotCold

Experiments 3 and 4: Cost Breakdown for 10 percent Write Probability

Costs in microseconds/Commit	ACBL		AACC		AOCC	
	Exp 3	Exp 4	Exp 3	Exp 4	Exp 3	Exp 4
Data Request	1659	1233	1504	1000	1405	1392
Write Lock Request	2000	1084	390	200	0	0
Client Application Processing	4047	2024	4047	2024	4400	2280
Commit	290	173	517	370	309	187

Figure 8.8: Experiments 3 and 4 Cost Breakdown

consistency algorithms. Faster CPUs help AACC and ACBL to reduce the CPU overhead associated with sending messages and they also help to reduce the execution time of a transaction. This, in turn, reduces the transaction blocking time in AACC and ACBL (due to locking conflicts) and increases overall throughput. Since AOCC has a higher abort rate than ACBL and AACC, with 50 percent abort variance, the restarted transactions in AOCC can access new pages which may not be present in either the client or the server buffer. This results in AOCC performing more I/Os per transaction than ACBL and AACC. Moreover, faster CPUs increase the disk utilization more quickly than slower CPUs, and this leads to higher relative disk I/O costs for AOCC than ACBL and AACC.

Figure 8.8 gives the cost breakdown for all of the three algorithms for both slow (experiment 3) and fast CPUs (experiment 4), respectively, for 10% write probability. The four costs presented in this figure are, 1) data request: the cost to obtain objects from the server (includes disk and network cost, and blocking related cost) and to put them in the client cache, 2) write-lock request: the cost for obtaining write locks from the server (includes blocking related cost), 3) client application processing: the cost for performing application related processing at the client (includes the aborted transaction processing cost) and 4) commit: the transaction commit processing cost. As seen in these figures, when going from slow CPUs to faster CPUs, the cost to get objects and locks from the server to the client decreases in AACC and ACBL because faster CPUs reduce the blocking overhead due to lock conflicts in these two algorithms. Faster CPUs also reduce the message transmission times in these two algorithms. As evident from a decrease in the client processing cost, faster CPU reduces the abort processing costs in AOCC. However, the decrease in the object request and lock request costs in AACC are larger in comparison to the decrease in the object request and client processing costs in AOCC. Thus, AACC is able to outperform AOCC. However, the synchronous nature of ACBL does not sufficiently reduce the locking costs (blocking overhead is still large) to allow ACBL to outperform AOCC.

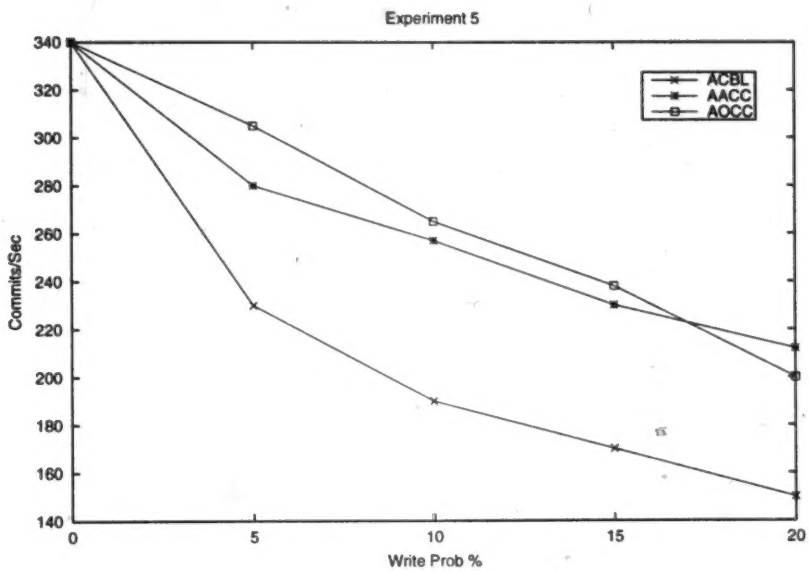


Figure 8.9: Zero Abort Variance

Experiment 5: Zero Abort Variance

The purpose of this experiment is to assess the impact of abort variance. It uses the same setup as experiment 4, except for the abort variance which has been set at 0%. As evident from Figure 8.9, AOCC outperforms ACBL and AACC because with no abort variance AOCC is able to find most of the data in the client cache during abort processing, reducing the number of data request messages to the server. This, in turn, ensures that it does not perform more disk I/Os than the other algorithms. As shown in Figure 8.10, the locking overhead present in AACC and ACBL allow AOCC to outperform them. However, as the write probability increases to 18%, AACC outperforms AOCC (there is a cross-over in the graph) because the number of aborts in AOCC is high, and the abort processing cost of AOCC starts to dominate, degrading its performance.

Experiment 5: Cost Breakdown for 10 percent Write Probability

Costs in microseconds/Commit	Algorithms		
	ACBL	AACC	AOCC
Data Request	1233	1000	1018
Write Lock Request	1084	200	0
Client Application Processing	2024	2024	2161
Commit	173	370	187

Figure 8.10: Zero Abort Variance Cost Breakdown

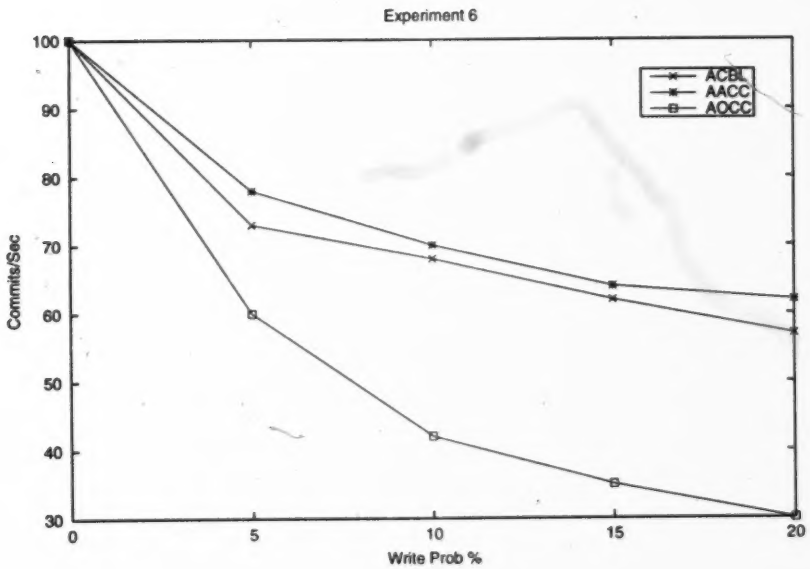


Figure 8.11: Small Server Buffer

Experiment 6: Small Server Buffer

The purpose of this experiment is to assess the impact of the combination of small server buffer and slow CPUs on the performance of the different algorithms. As shown in Experiment 3, slow CPUs allow AOCC to slightly outperform AACC, and this experiment is trying to assess whether small server buffers overturn this result. This represents the case when there is contention for the server buffer due to simultaneous access by many clients. This experiment's setup is the same as Experiment 3 (slow CPUs), except for the small server buffer. This experiment uses an abort variance of 50%. Figure 8.11 shows that AACC and ACBL outperform AOCC even though this experiment is using slow CPU speeds. Small server buffer leads to more misses at the server cache, and this leads to higher contention at the server disks. With 50% abort variance, AOCC requests more objects from the server, but the small server buffer leads to more misses at the server buffer. This allows even ACBL to outperform AOCC due to the higher restart processing costs of AOCC.

Experiment 7: Small Server Buffer and 0% Abort Variance

The purpose of this experiment is to assess whether 0% abort variance changes the results of Experiment 6. This experiment's setup is the same as experiment 6, except for the abort variance of 0%. As shown in Figure 8.12, AOCC's performance matches the performance of the other two algorithms. Similar to the findings in Experiment 5, a 0% abort variance helps AOCC to not perform more disk I/Os than the other algorithms. Therefore, higher disk contention due to smaller server buffer affects all the three algorithms equally. AACC slightly outperforms AOCC as the write probability reaches 18% because of the high abort rate in AOCC.

Experiment 8: Small Client Log Buffer

The purpose of this experiment is to assess the impact of a small client log buffer on the performance of the three algorithms. This experiment concen-

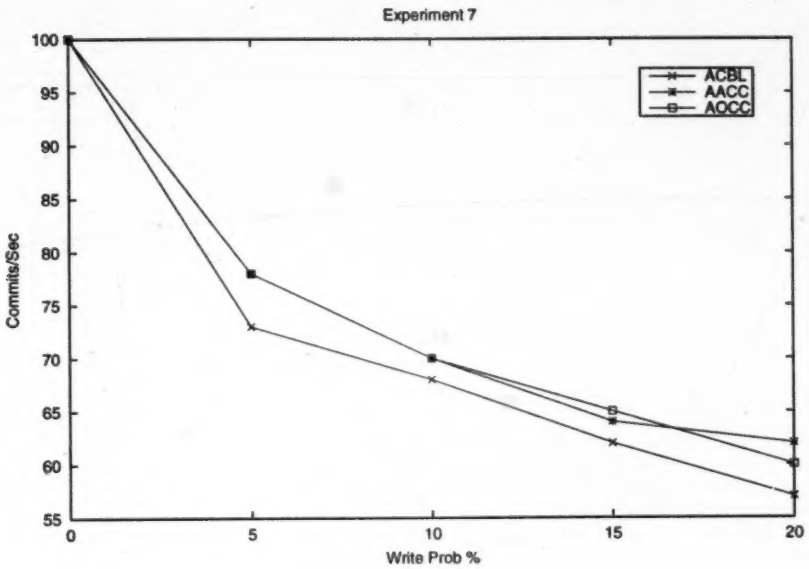


Figure 8.12: Small Server Buffer and 0% Abort Variance

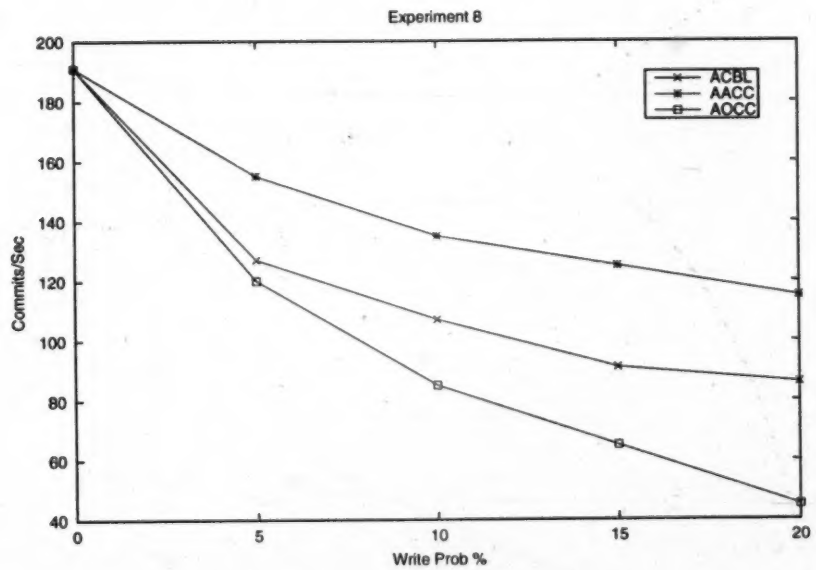


Figure 8.13: Small Client Log Buffer

trates on the impact of a small client log buffer which occurs if (1) if the log buffer is shared by multiple client processes, or (2) if one is dealing with a transaction that accesses and updates many objects, or (3) large portions of large objects are updated. With a small client log buffer, it is not possible for the client to keep all of the transaction undo logs in the log buffer. The setup of this experiment is similar to experiment 3, except that the client log buffer size has been set to zero and the abort variance has been set to 0%. As shown in Figure 8.13, AACC and ACBL outperform AOCC. With a small client log buffer, AOCC is not able to store all of the undo logs in the client buffer. Therefore, during transaction abort processing, the client has to re-request the data from the server and this increases the transaction execution cost.

A small client data buffer degrades the performance of not only AOCC (due to higher abort processing costs of re-acquiring the data from the server), but also the performance of AACC and ACBL (because transactions block for a longer time during lock conflicts). However, having a small client log buffer hurts AOCC more, since the abort rates of ACBL and AACC are quite low and they rely less on the client undo log buffer.

Experiment 9: Fast Network

The purpose of this experiment is to see whether a fast network helps to overturn the results presented in Experiment 3. That is, slow CPU speed benefits AOCC, and this experiment is trying to assess whether a fast network helps AACC and ACBL to offset this benefit. Similar to experiment 3, this experiment uses slow CPU speeds with 50% abort variance, and a large server buffer. However, a faster network (155 Mbps) is used. As shown in Figure 8.14, a faster network helps the performance of all three algorithms. AACC outperforms both AOCC and ACBL, and AOCC outperforms ACBL when write probability is less than 18% because its abort rate processing is still not high enough to allow ACBL to match its performance. However, as the write probability increases, the high abort rate of AOCC degrades AOCC performance, and ACBL is able to match and beat AOCC's performance.

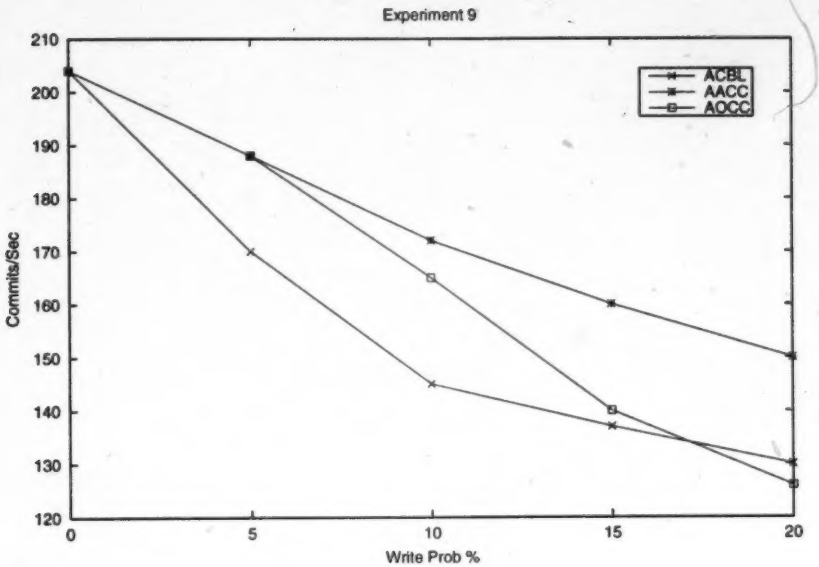


Figure 8.14: Fast Network

Experiment 10: Slow Network

The purpose of this experiment is to assess whether slow networks neutralize the performance advantages realized by AACC and ACBL due to fast CPUs. Since AACC and ACBL send more messages than AOCC, this experiment tries to assess the impact of slow networks. This experiment's setup is the opposite of experiment 9 in that it uses a combination of slow network (10Mbps) and fast CPUs. As seen in Figure 8.15, the performance of AOCC, and ACBL are quite similar. However, AACC outperforms AOCC because the latter continues to incur higher disk overhead due to the combination of higher abort rate and the presence of 50% abort variance in the workload. AACC outperforms ACBL because the latter incurs message blocking overhead due to the use of synchronous messages. AOCC's performance is identical to ACBL's performance from 10% write probability onwards because the abort processing cost in AOCC degrades its performance. In slow and congested networks it costs AOCC more to send pages from the server to the client during abort processing.

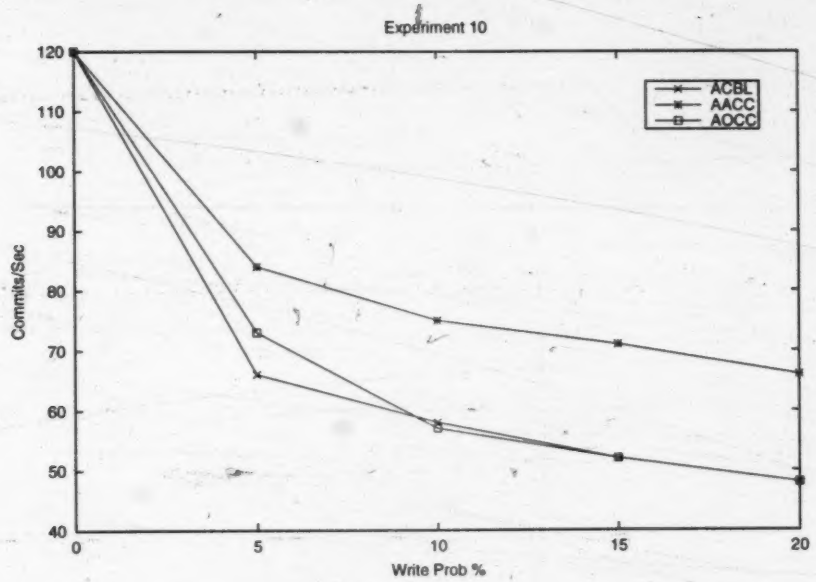


Figure 8.15: Slow Network

Experiment 11: Fast Disks

The purpose of this experiment is to assess whether faster disks help AOCC to outperform AACC in the presence of fast CPUs. Thus, it uses the setup of Experiment 4, but uses fast disks. The speeds of the disks (described in Chapter 7) has been doubled in this experiment. Fast disks help reduce the server disk utilization from 75% to around 35%. As shown in Figure 8.16, AACC is still able to outperform AOCC and ACBL. In comparison to Experiment 4 (fast CPUs) where AOCC loses to AACC, its performance is close to AOCC performance for up to 10% write probability. However, AOCC loses to AACC as the write probability increases due to the higher abort processing overhead in AOCC. But the gap between AACC and AOCC is narrower in this experiment than in Experiment 4. Since AOCC incurs a higher number of aborts, during abort processing, it performs disk I/Os in order to retrieve the pages which are not present in the server cache. Therefore, faster disk speed helps AOCC more than the other algorithms. However, the gains are not sufficient to change the performance order between the three

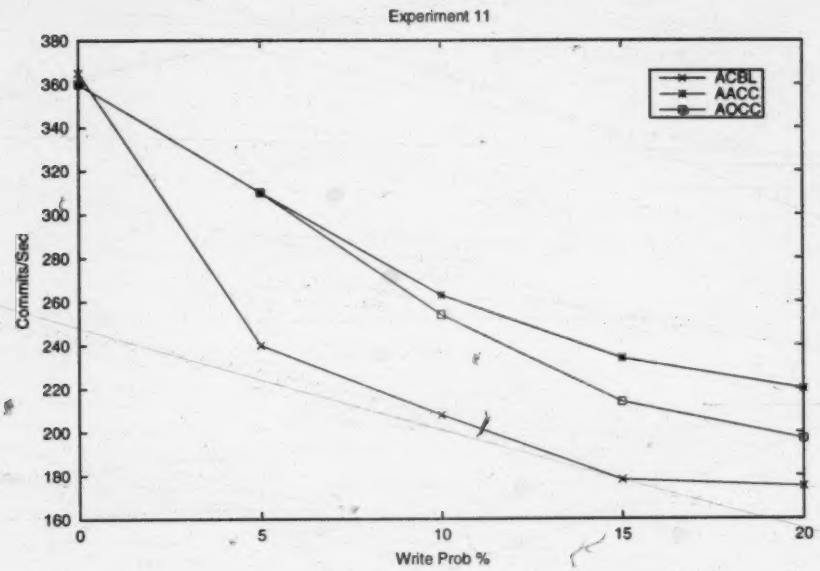


Figure 8.16: Fast Disks

algorithms.

Experiment 12: Small Client Buffer

The purpose of this experiment is to assess the impact of high network and server CPU contention on the performance of the different algorithms. Network and the server CPU can get saturated when the number of clients that are simultaneously accessing the server is large. In this experiment setup, the server buffer is large to ensure that there is no disk contention. The client buffer has been set to be smaller than the client working set size to ensure that there are many misses at the client cache and therefore, there is network contention (due to many simultaneous client requests from the different clients) and server CPU contention (due to a large amount of message processing overhead). The client cache size has been set to be 20% of a client's hot region. The server CPU utilization is around 75% and the network utilization is near 90%. As can be seen in Figure 8.17, AOCC outperforms the other algorithms as the write probability is lower than 18% due to lower messaging and blocking costs.

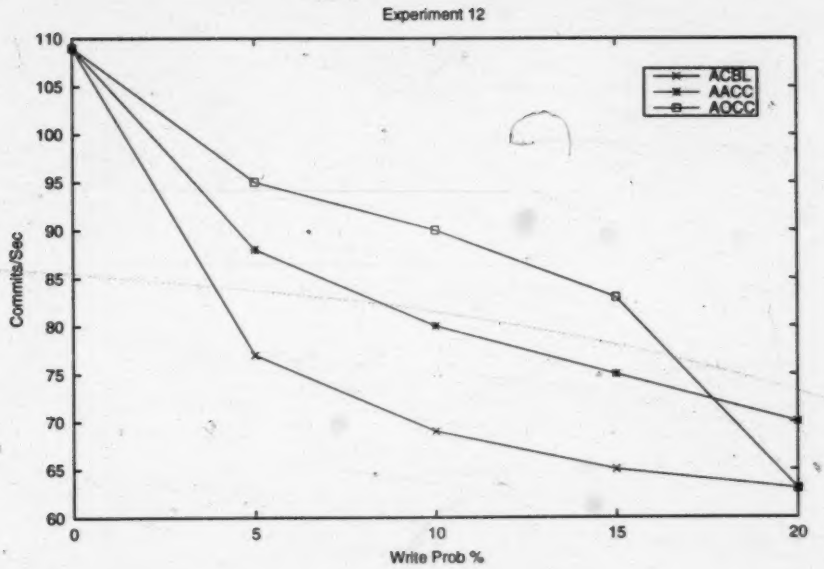


Figure 8.17: Small Client Buffer

Furthermore, the lack of server disk contention also enables AOCC to have low abort processing costs. Network contention and server CPU contention affect ACBL more than AACC because in ACBL clients issue synchronous lock escalation messages and they block until they receive lock responses back from the server. Thus, ACBL's performance trails that of AACC. Network and server CPU contention affect AACC more than they affect AOCC because AACC issues more number of messages than AOCC. AACC starts to outperform AOCC when the write probability reaches 18% because the number of aborts in AOCC increases with an increase in the write probability. ACBL's performance matches AOCC's performance when the write probability reaches 18% because of higher abort processing costs in AOCC.

Experiment 13: High Spatial Locality

The purpose of this experiment is to assess the impact of the combination of high spatial locality and Sh-HotCold workload on the three algorithms. The spatial locality percentage has been set to 70%. The setup of this experiment

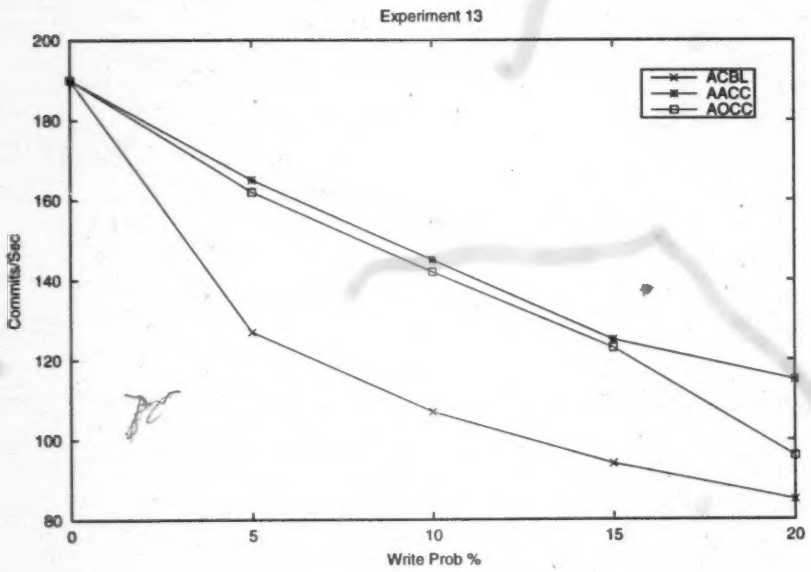


Figure 8.18: High Spatial Locality

is the same as Experiment 3. As shown in Figure 8.18, the performances of AOCC and AACC are quite similar. When the write probability approaches 18%, AACC outperforms AOCC, because of the latter's higher abort rate. Therefore, higher spatial locality has not changed the relative ordering of the performance between AOCC and AACC. Both AOCC and AACC outperform ACBL because ACBL has higher messaging and message blocking overhead. High spatial locality leads to fewer pages in a client's working set. This helps ACBL, but it is still not enough to overcome the presence of message blocking overhead in ACBL.

Experiment 14: 50 Percent Server Work Allocation

The purpose of the server allocation workload is to assess the impact of abort processing overhead for AOCC when work is partly performed at the server. The server and the clients have large buffers and they use fast CPUs. This experiment uses 100 Mbps network and 50% abort variance. Sh-HotCold workload is used both at the server and at the clients. 50% of the work is performed

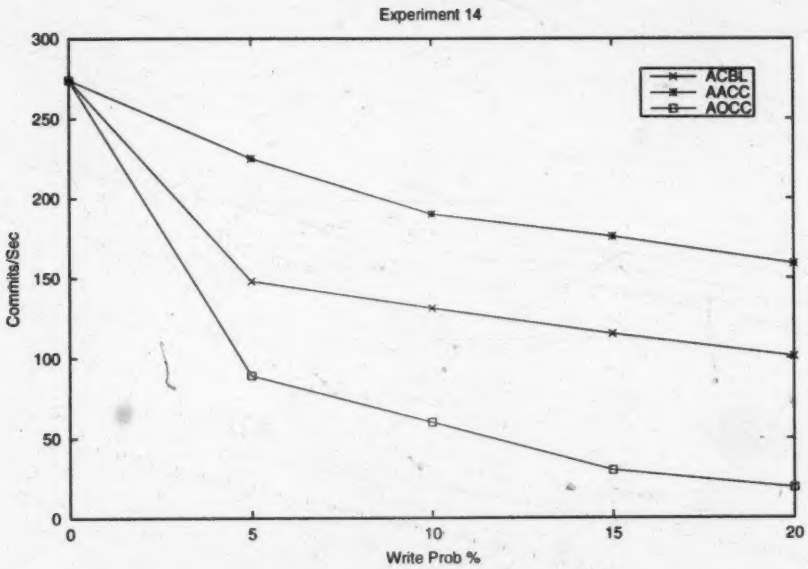


Figure 8.19: 50% Server Work Allocation

at the server and 50% of the work is performed at the clients. As can be seen in Figure 8.19, AACC and ACBL outperform AOCC as the write probability increases because AOCC incurs more transaction aborts. In the environment where the work is strictly performed at the client, if a transaction aborts then the re-execution of the failed transaction has very little impact on the performance of the other transactions. However, in this experiment, since application processing is also performed at the server, when a transaction aborts it also impacts the performance of other client transactions. The key reason is that server resources such as the CPU, buffers, data disks and log disk are shared by all of the clients. Therefore, during the transaction abort processing, the necessary data pages and logs might not be present in the server buffer and have to be retrieved from disk. This not only slows down the abort processing of the failed client transaction, but also degrades the throughput of the entire system. Since AACC and ACBL are avoidance-based, they incur fewer aborts than AOCC, and hence are able to outperform AOCC. Previously, it was shown within the context of centralized DBMSs that in medium to highly contended servers, the performance of optimistic concurrency control

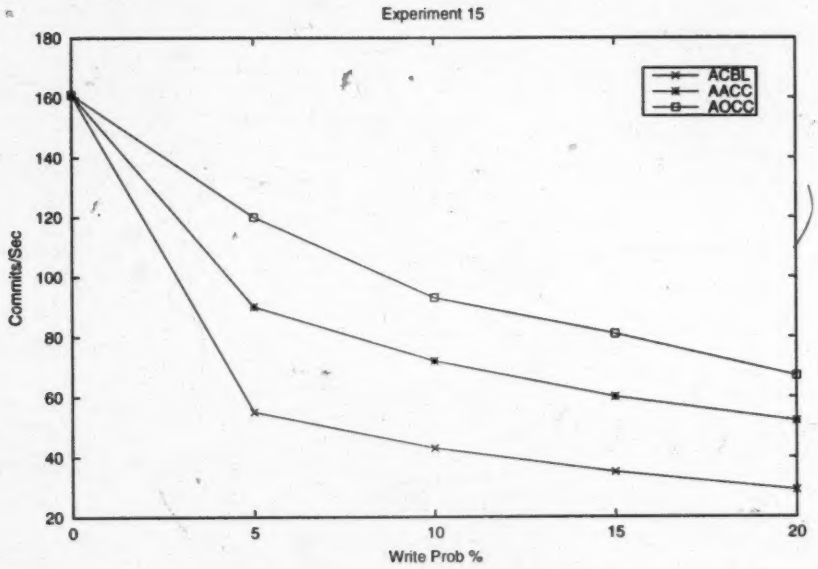


Figure 8.20: Network Delay

algorithms suffers due to high abort processing costs [ACL87]. The results of this experiment concur with the assessment of the previous study.

Experiment 15: Network Delay

This experiment assesses the impact of network delays (such as those experienced in wide area network environments and presented in Figure 7.2) on the performance of the three algorithms. This experiment uses fast CPUs, 100 Mbps network, large client and server buffers and an abort variance of 50%. Figure 8.20 shows that the performance of the three algorithms degrades, for the Sh-Hotcold workload, as the network delay is introduced in comparison to a network with no delays (Figure 8.7). However, the performance of ACBL degrades much more (percentage-wise) than the performance of AOCC and AACC, because ACBL uses synchronous lock escalation messages whereas AACC and AOCC use asynchronous and deferred lock escalation messages, respectively. In ACBL, the clients remain blocked until their lock escalation and subsequent callback messages (if necessary) are processed. AOCC out-

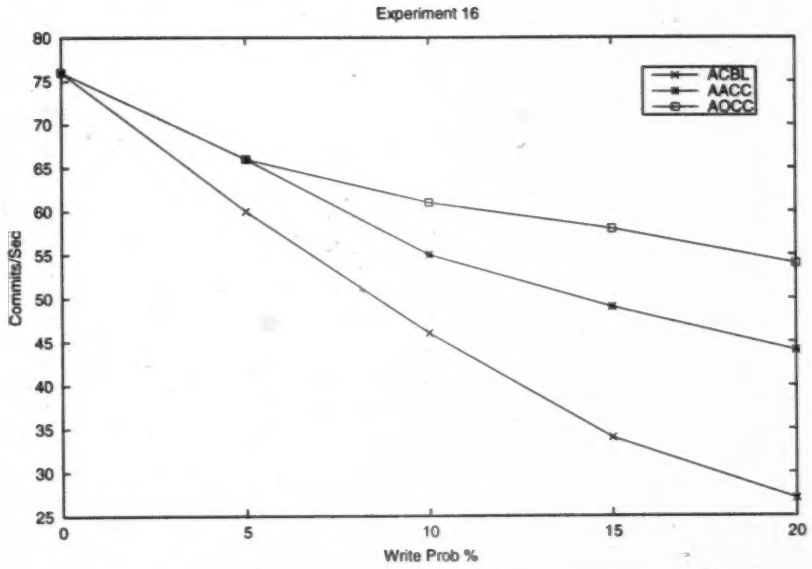


Figure 8.21: HiCon Workload

performs AACC because the latter uses more messages than AOCC. Even though AACC uses asynchronous lock escalation messages, a client's transaction synchronizes with other clients at commit time to ensure that there are no conflicts. Thus, unexpected message delays increase the commit processing time in AACC, and its performance trails AOCC's performance.

8.1.4 HiCon Workload

In HiCon workload, the clients access the shared data region 80% of the time and the data region of other clients 20% of the time. This is a skewed data access pattern that is not usually present in data-shipping applications [CFZ94]. It is being examined here to test the behavior of the different cache consistency algorithms under extreme data contention situations.

Experiment 16: HiCon Data Contention

This experiment uses 100 Mbps network, 50% abort variance, and small server buffer and fast CPU speeds. In the previous experiments, all of these factors

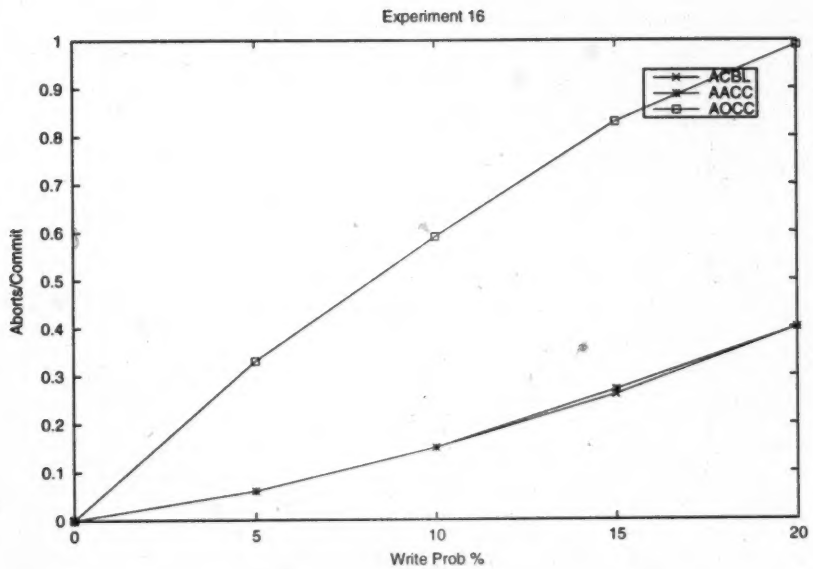


Figure 8.22: HiCon Abort Rate

Experiment 16: Cost Breakdown for 10 percent Write Probability

Costs in microseconds/Commit	Algorithms		
	ACBL	AACC	AOCC
Data Request	11780	11770	11398
Write Lock Request	7863	805	0
Client Application Processing	2306	2295	3055
Commit	160	1350	187

Figure 8.23: HiCon Costs

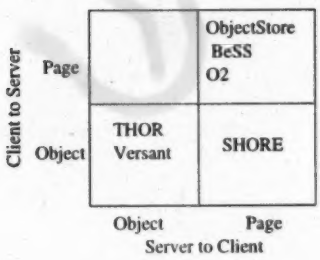


Figure 8.24: ODBMS Classification According to Data Transfer

helped AACC and ACBL more than AOCC. Therefore, the objective of this setup is to see the impact of the HiCon data sharing pattern on the three algorithms. As shown in Figure 8.21, even with faster CPUs and 50% abort variance, AOCC outperforms AACC which outperforms ACBL. However, as shown in Figure 8.22, AOCC has a higher abort rate (aborts/commits) than ACBL and AACC. One would expect algorithms with a high abort rate to perform worse than algorithms with lower abort rates. As described in Chapter 6, the read/write conflict blocking rates of AACC and ACBL are higher than the abort rate of AOCC. That is, for every blocking transaction in AACC and ACBL, the equivalent situation can lead to either an abort or a commit in AOCC. As shown in Figure 8.23, the time a transaction remains blocked in ACBL and AACC (higher object request and write lock request costs) is more than the abort processing cost in AOCC. Thus, AOCC outperforms ACBL and AACC even though it encounters a higher abort rate.

8.2 Integrated Performance Study

The hybrid server (HybSrv) architecture proposed in this dissertation is compared with a software-based page server (PageSoft), a hardware-based page server (PageHard), and an object server (ObjSrv). The software-based page server falls under the Page-Object server classification of Figure 8.24 and is similar to SHORE [CDF+94]. The hardware-based page server falls under the

	Server->Client	Recovery Client->Server	Cache Consistency	Pointer-Swizzling	Server Buffer	Client Buffer
PageHard	Page	ARIES Logs and Page	Page Level AACC	Hardware	Page/ Modified Page	Page
PageSoft	Page	Redo At Server Logs	Adaptive AACC	Software	Page/ Modified Object	Dual
ObjSrv	Objects	Redo At Server Logs	Adaptive AACC	Software	Page/ Modified Object	Object
HybSrv	Page / Objects	ARIES Or Redo at Server Page/Logs	Adaptive AACC	Software	Page/ Modified Dual	Dual

Figure 8.25: Systems Under Comparison

Page-Page server classification, and is similar to ObjectStore [LLOW91] and BeSS [BP95] in that it sends pages in both directions during client-server interaction. The object server architecture falls under the Object-Object server classification and is similar to Versant [Ver98] and Thor [LAC+96]. The existing hardware page server systems [BP95, LLOW91] employ page level data transfer, concurrency control and buffer management. As a representative of these systems, PageHard also adheres to the page level restrictions and this is the key distinguishing feature between PageHard and the other architectures. The data transfer mechanism from the server to the client is the key distinguishing factor between PageSoft and ObjSrv. The ability to send pages or objects from the server to the client, and to return pages or objects from the client are the key distinguishing factors between HybSrv and the other architectures (PageSoft, ObjSrv and PageHard). The overall performance of a system is also affected by other issues such as query processing, query optimization, indexing and others, which are not considered here. The latest advances in cache consistency, buffer management, and recovery strategies are incorporated into all of the systems under comparison in this study (see Figure 8.25), ensuring that they all benefit from the same advantages. Therefore, the systems under comparison are similar, but not identical to their commercial/research counterparts. In the remainder of this section, the details of these architectures are described.

Hardware-Based Page Server (PageHard)

In the hardware-based page server architecture, the client requests a page from the server by sending it the page identifier. The server responds to the request by returning the appropriate disk page to the client. This architecture uses the hardware-based pointer swizzling mechanism as in ObjectStore [LLOW91] and BeSS [BP95]. Since the hardware-based pointer swizzling mechanism relies on the operating system virtual memory faulting (page level) mechanism, it is efficient for the clients to only deal with pages. This, in turn, makes it necessary for the server to return pages to the clients, and for the clients to manage a page level data buffer. The server manages a page level staging read buffer and a modified page buffer (MPB), which is a page level version of MOB. Even though ObjectStore and BeSS use the ACBL cache consistency mechanism, PageHard uses AACC because of its better performance. A hardware-based pointer swizzling system relies on the operating system provided page protection mechanism for read and write lock support. This makes it difficult to provide object level locking support, and, thus, PageHard uses the page-level version of AACC. PageHard (like BeSS) uses the ARIES recovery algorithm, because a previous study [WD95] has shown that the log disk becomes a bottleneck during whole-page logging, which is used in ObjectStore. Similar to ObjectStore and BeSS, at commit time the clients return updated pages back to the server. In this architecture the clients maintain a page level undo log buffer. Log records are generated by the client by performing a *page difference* operation [WD95], and they are stored by the server in a log buffer from where they are flushed to the log disk when the buffer is full or when the transaction has reached the commit point. The data buffers use the LRU like (second chance) buffer replacement policy and the log buffers use the FIFO buffer replacement policy. PageHard uses 8 byte pointers to represent OIDs because using 4 byte pointers limits the amount of addressable virtual memory to 4 gigabytes, and this, in turn, restricts the size of the database that can be accessed by a client.

Software-based Page Server (PageSoft)

In the software-based page server architecture the client request and server response are identical to PageHard. PageSoft uses software pointer swizzling mechanism and it uses LOIDs that are 8 bytes long. Since the software pointer swizzling mechanism does not use the operating system page faulting mechanism to load data, the clients have the flexibility to manipulate both pages and objects. Thus, this architecture provides both page level and object level concurrency control support. PageSoft also uses AACC because of its superior performance. In SHORE, the clients receive pages from the server but then the relevant objects are copied from the page buffer into an object buffer before an application can access them. In order to reduce this copying overhead, PageSoft uses a hybrid dual buffer at the clients [KK94]. The dual buffer allows clients to store well clustered pages as well as isolated objects from badly clustered pages. Similar to SHORE, PageSoft utilizes the redo-at-server logging mechanism. However, unlike SHORE, which does not contain a MOB, PageSoft contains both a staging read buffer and a MOB. The clients maintain an object level log buffer and generate log records using the *difference* operation. The server also maintains a staging log buffer from where log records are flushed to the log disk either when the log buffer gets full or at commit points. In this architecture, the data buffers use the LRU like (second chance) buffer replacement policy and the log buffers use the FIFO buffer replacement policy.

Grouped Object Server (ObjSrv)

In this architecture, the client requests objects by sending object identifiers to the server. The server returns a group of objects to satisfy the client's object request. This data transfer mechanism is similar to Versant ODBMS [Ver98] and the THOR storage manager [LAC+96]. Similar to THOR [LAC+96], in this architecture the clients dynamically determine the size of the object group and the server forms object groups using objects that reside on contiguous disk locations. The server contains a staging read buffer and a MOB buffer. The

server also contains a log staging buffer for writing the log records to the log disk. Since the clients receive a group of objects, they must maintain an object level buffer. Furthermore, they cannot return updated pages back to the server because the clients only deal with objects. Since clients in object server do not deal with pages, the clients cannot efficiently use the hardware pointer swizzling mechanism [WD95] and, therefore, they use the software pointer swizzling mechanism. This architecture utilizes the redo-at-server recovery mechanism. Similar to THOR, the server stores the redo logs in its MOB. Unlike THOR, which employs an optimistic cache consistency mechanism (AOCC), this architecture uses the object server version of AACC which has been shown to be more robust [ÖVU98]. In this architecture, the data buffers use the LRU like (second chance) buffer replacement policy and the log buffers use the FIFO buffer replacement policy.

Hybrid server (HybSrv)

In this new architecture that is proposed in the dissertation, the clients can request either pages or objects from the server, and the server can return either pages or groups of objects to the clients. The clients can also return both updated pages and objects to the server. When a client returns updated objects, it uses a redo-at-server recovery mechanism, and when it returns updated pages, it uses ARIES-CSA type recovery mechanism. HybSrv uses software pointer swizzling, since it has to efficiently handle both pages and objects. Since the clients can receive either pages or objects, they maintain a dual buffer. The server maintains a staging read buffer, as well as a modified dual (page/object) buffer. The clients maintain an object level log buffer and generate log records using the *difference* operation. The server also maintains a staging log buffer from where log records are flushed to the log disk either when the log buffer gets full or at commit points. In this architecture, the data buffers use the LRU like (second chance) buffer replacement policy and the log buffers use the FIFO buffer replacement policy.

8.2.1 Integrated Study Outline

The following parameters are varied in this performance study:

- **Buffer Size:** The client and the server buffer configuration is the primary system parameter. The sizes of the client and server buffers have a major impact on the client and the server cache miss rates. Client cache management has an impact on the number of data requests made from the client to the server. The server buffer management has an impact on the number of disk I/Os performed at the server. The following four client-server buffer configurations are used in this study:

  - **Small-Small:** In this configuration both the client and the server buffers are small. A small client buffer means that the client working set does not fit into the client cache. A small server buffer means that the combined working sets of the clients do not fit into the server buffer and the server disk utilization is higher due to server buffer misses. Small client buffer scenario is possible if the client cache is shared by multiple client processes. A small server buffer scenario is possible if multiple clients are simultaneously accessing the server. It is important to note that the relative size of the buffers with respect to the working sets is more important rather than the absolute buffer sizes [AGLM95, CFZ94]. It would have been preferable to model the small server buffer case by keeping the server buffer size constant and by increasing the number of clients. However, the memory constraints of the simulator did not allow for this type of modeling. Reducing the server buffer size captures the essence of the impact of too many clients on the server buffer.

  - **Small-Large:** In this configuration the client buffer is small but the server buffer is large. A small client buffer has the same meaning as above, whereas a large server buffer means that the clients cumulative working sets fit in the server cache. A large server buffer scenario is possible if not too many clients are simultaneously ac-

cessing the server, or if the clients are accessing the same region of the database.

  - **Large-Small:** In this configuration the client buffer is large and the server buffer is small. A large client buffer means that the client's working set fits into the client buffer. A small server buffer has the same meaning as described above. A large client buffer is possible if the client has a lot of memory (and thus a large cache), and the client working set is relatively small.

  - **Large-Large:** In this configuration the client buffer and the server buffer are large. A large client and server buffer have the same meanings as described above.

- **Data Clustering Probability:** Data clustering is a key workload parameter which determines whether it is beneficial to transfer a page from the server to the client and whether it is beneficial to cache a page at the client. Temporal locality, access locality and spatial locality are the key data clustering parameters that are varied and are specified separately for each experiment.

- **Object Write Probability:** The object write probability has an impact on read-write and write-write conflicts. The object write probability also determines whether it is beneficial to return updated pages or objects from the clients to the server. The object write probability is varied between 0 and 20%. The update probability of most applications does not exceed 20% [CK89, Ghe95].

- **Data Sharing Pattern:** *Private* and *Sh-HotCold* are the two data sharing patterns used in this experiment study.

- **Network Speed:** Network speed plays an important role as it determines whether sending badly clustered pages degrades performance. This study uses slow, normal and fast network speeds. The slow speed corresponds to 10Mbps network, 10000 cycles/message fixed CPU cost

and 7 cycles/byte message variable CPU cost. The normal speed corresponds to 100 Mbps network, 6000 cycles/message fixed CPU cost and 4 cycles/byte message variable CPU cost. The fast speed corresponds to 155 Mbps network, 2000 cycles/message fixed CPU cost and 2 cycles/byte message variable CPU cost.

- **Page Size:** Page sizes of 4K, and 16K are used in this dissertation.

8.2.2 Large Client and Large Server Buffers

The purpose of this experiment is to assess the impact of pointer swizzling mechanism and client to server data transfer mechanism on the overall performance of the different architectures. In this setup both the client and the server have large buffers. In steady state, the client cache is loaded, and, therefore, there should be few client cache misses. Due to these conditions, buffer management and server-to-client data transfer are not the performance differentiating factors between the different architectures. Instead, pointer access and client-to-server data transfer are the key performance determining issues. The client buffer is large enough to hold the client working set and the server buffer is 75% of the database size. There are three pointers from each object to other objects. Both the spatial and temporal locality have been set at 50% and the network speed is set to 100 Mbps.

Experiment 17: Large-Large Private

This experiment uses the private data sharing pattern. Therefore, concurrency control and cache consistency are not an issue in this experiment. Write probability is varied on the x-axis and the overall system throughput in commits per second is measured. As seen in Figure 8.26, PageHard is outperformed by all of the architectures for write probabilities greater than zero. However, there is no difference between the performance of the different architectures that are using software pointer swizzling mechanism. Moreover, as shown in Figure 8.27, the pointer swizzling related costs are small in comparison to

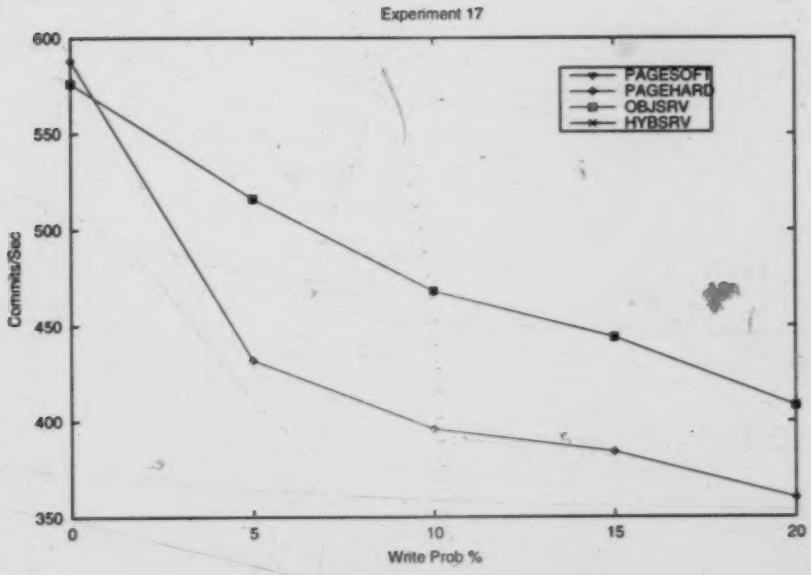


Figure 8.26: Large-Large Buffer Setup

Experiment 17: Cost Breakdown for 10 percent Write Probability

Costs in microseconds/Commit	Algorithms			
	PageHard	PageSoft	ObjSrv	HybSrv
Read Cost	16500	16500	16500	16500
Write Cost	7280	7280	7280	7280
Client to Server Data Transfer Cost	5665	1155	1155	1155
Pointer Swizzling Cost	0	249	249	249

Figure 8.27: Large-Large Buffer Setup: Private Workload

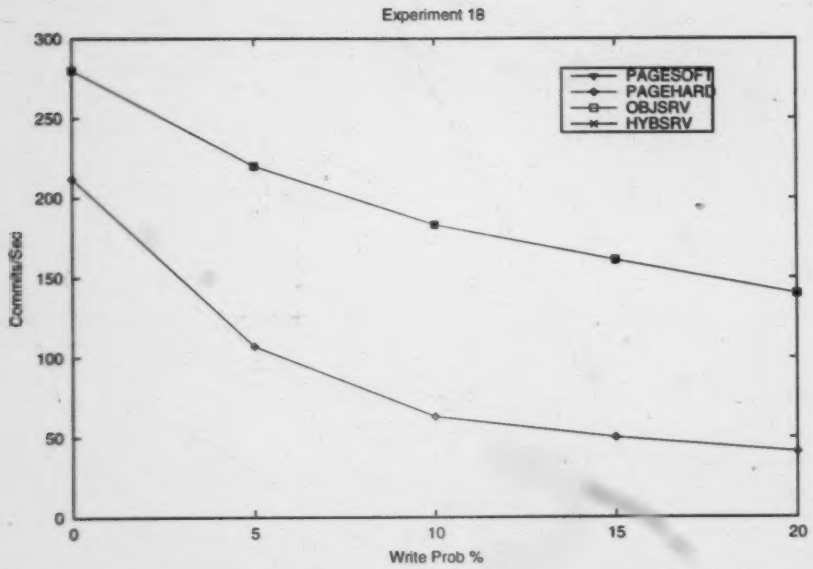


Figure 8.28: Large-Large: Sh-HotCold Workload

other costs. Therefore, pointer swizzling costs are not a major component in determining the overall performance in environments where the client cache can hold the entire client working set and the applications are performing some processing. For 0% write probability, PageHard slightly outperforms the other architectures because it does not encounter pointer indirection cost. However, as the write probability increases, the other architectures outperform PageHard, because they send updated objects to the server, whereas, PageHard sends sparsely updated pages to the server and encounters higher communication overhead.

Experiment 18: Large-Large Sh-HotCold

The purpose of this experiment is to test the impact of write data sharing on the performance the different algorithms. The experiment setup used here is the same as experiment 17 setup. However, this experiment uses the Sh-HotCold workload. As seen in Figure 8.28, PageHard is outperformed by all of the other algorithms, because it manages concurrency control strictly at page

level and, therefore, it does not allow multiple clients to simultaneously read and write to different portions of a locally cached page. The use of operating system provided page level access protection mechanisms makes it difficult for PageHard to provide object level concurrency control, and, thus, the pointer swizzling benefits of PageHard are lost.

Another important result of this experiment, as shown in Figure 8.28, is that ObjSrv and HybSrv are able to compete with PageSoft. Previously it was thought that systems that transfer data at object level cannot implement coarse-grained concurrency control mechanisms [DFMV90, CFZ94]. However, the concurrency control enhancement proposed for ObjSrv and HybSrv in this dissertation allow them to efficiently use a non-optimistic cache consistency/concurrency control mechanism.

8.2.3 Small Client and Large Server Buffers

In this system configuration (referred to as Small/Large) the client's working set does not fit into its cache even if the client has a lot of physical memory. This is possible if the size of the working set is very large or if the client buffer is shared by multiple applications. The client buffer is 1.5% of the database size and the server buffer is still 75% of the database size. The primary goal of this configuration is to compare the server to client data transfer mechanisms. Network speed, page size, and clustering are varied for this buffer setup to assess the robustness of the different data transfer methods.

Experiment 19: Small-Large with Good Access Locality

This experiment uses private workload with 10% write probability. The spatial locality has been varied to see the relationship between clustering and client buffer size. Temporal locality has been set at 50% and access locality has been set at 90%. As shown in Figure 8.29, PageHard performs the worst during low spatial locality because it manages the client cache strictly at page level and this leads to low client buffer utilization. The clients in ObjSrv, HybSrv and PageSoft only retain useful objects in their cache. As shown in

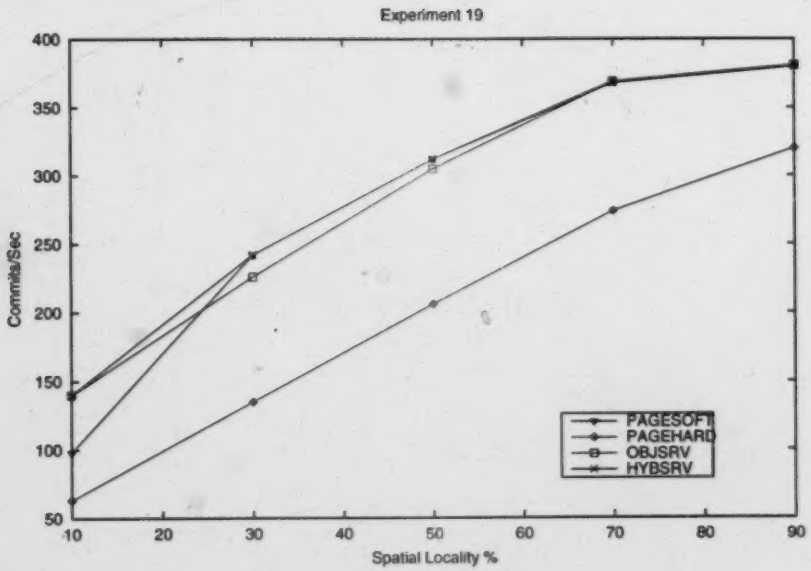


Figure 8.29: Small-Large Good Access Locality

Experiment 19

	Algorithms							
	PageHard		ObjSrv		PageSoft		HybSrv	
	10%	30%	10%	30%	10%	30%	10%	30%
Client Cache Misses Per Commit	33.9	8.9	26.4	10.3	25.2	7.9	25.3	7.9
Server To Client Data Transfer Cost in Microseconds Per Commit	9409	3167	3070	1434	6393	1076	1072	1077

Figure 8.30: Good Access Locality Cost Breakdown

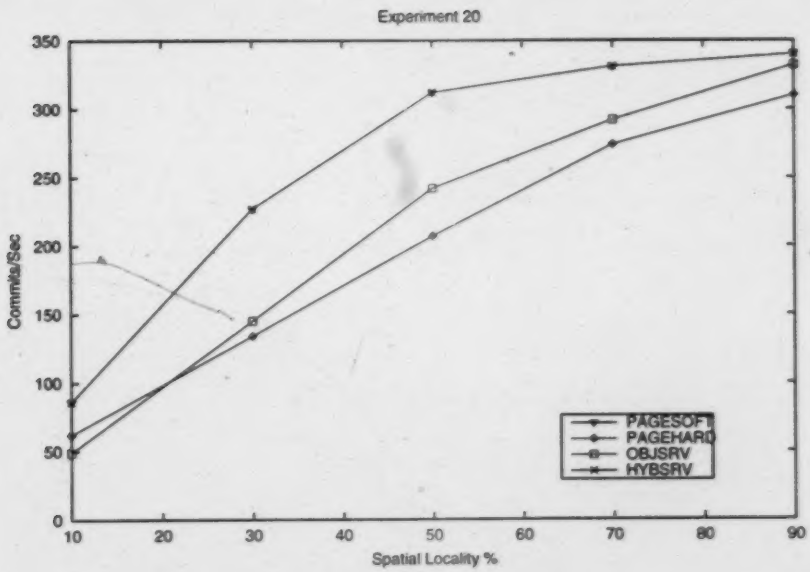


Figure 8.31: Small-Large Bad Access Locality

Figure 8.30, the low client buffer utilization in PageHard leads to a higher number of client cache misses and this, in turn, degrades PageHard's performance. Returning of updated pages instead of updated objects increases the network overhead encountered by PageHard and this also contributes towards the lower performance of PageHard. Since PageHard employs the hardware pointer swizzling mechanism, the cost of loading pages into the client cache is higher for PageHard than for the architectures employing the software pointer swizzling mechanisms.

The second important result is that ObjSrv and HybSrv outperform PageSoft during bad spatial locality, because, as shown in Figure 8.30, PageSoft encounters higher network overhead as a result of sending badly clustered pages from the server to the client. Since HybSrv is able to switch and operate as an object server, its performance is better than the architectures that send badly clustered pages to the clients.

Experiment 19

Algorithms

	PageHard	ObjSrv	PageSoft	HybSrv
Client Cache Misses Per Commit	34.0	107.0	28.9	29.0
Server To Client Data Transfer Cost in Microseconds Per Commit	9088	9226	7368	7363

Figure 8.32: Bad Access Locality Cost Breakdown

Experiment 20: Small-Large with Bad Access Locality

The purpose of this experiment is to assess whether sending a group of objects is beneficial when the access locality is bad. The setup of this experiment is similar to Experiment 19. However, in this case, the access locality has been set to 10%. As seen in Figure 8.31, PageSoft and HybSrv outperform PageHard and ObjSrv. PageHard is outperformed due to the higher cost of loading pages into the client cache (due to hardware pointer swizzling) and due to returning updated pages to the server. ObjSrv is outperformed by PageSoft because in this workload the access locality is bad, causing the ObjSrv server grouping mechanism to be less accurate. During bad access locality, multiple non-contiguous objects on a page are accessed together, and due to the inaccuracy of the object grouping mechanism, clients in ObjSrv have to make multiple data requests to the server. As shown in Figure 8.32, ObjSrv encounters a higher number of misses in the client cache, and, therefore, as shown in Figure 8.32, it sends more data requests to the server (incurs greater network overhead). Since HybSrv switches over to sending pages during periods of bad access locality, it outperforms ObjSrv. During periods of bad access and bad spatial locality, it is better to send pages to the clients and let the client dual buffering mechanism retain only useful objects in the client cache.

Thus, even during bad spatial locality, if the access locality is bad, it is desirable to send pages from the server to the client because storing pages improves the client cache hit rate. Experiments 19 and 20 are important

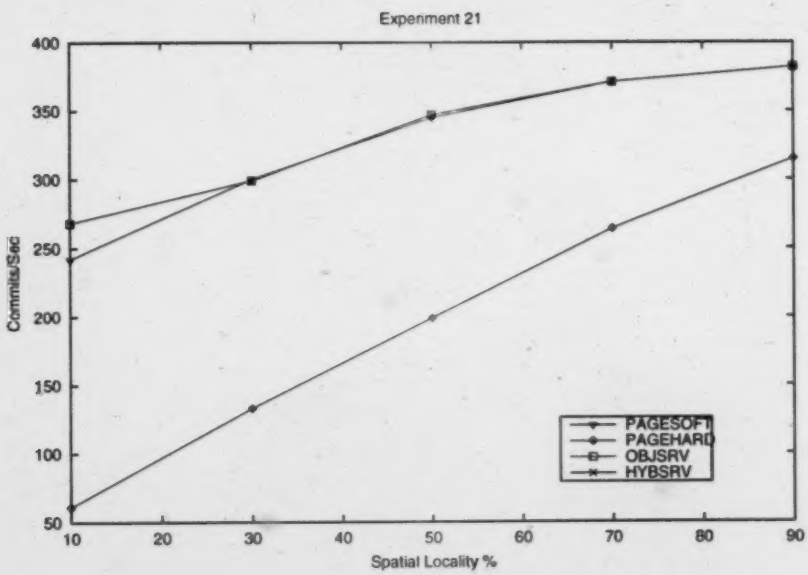


Figure 8.33: Small-Large High Temporal Locality

because, unlike a previous study comparing grouped object servers and page servers [LAC+96], the results of these experiments show that the ObjSrv object grouping mechanism does not always outperform the architectures that send pages from the server to the client. Similarly, the page server architecture using a dual buffer at the client does not always outperform the grouped object server approach. These two experiments justify the need for an adaptive hybrid server to client data transfer approach.

Experiment 21: High Temporal Locality

The purpose of this experiment is to assess the impact of high temporal locality. Therefore, Experiment 19 setup with a temporal locality of 90% is used here. As shown in Figure 8.33, ObjSrv and HybSrv outperform PageSoft when the spatial locality is 10% because PageSoft transfers badly clustered pages from the server to the clients. PageHard is outperformed by all the algorithms for all of the spatial localities because it transfers badly clustered pages, and also because it manages the client buffer strictly at page level. However, unlike

in Experiment 19, as the spatial locality improves, ObjSrv is able to compete with PageSoft because with high temporal and spatial localities, a group of contiguously located objects are repeatedly accessed with high probability. This, in turn, allows the server-based object grouping algorithm that forms object groups consisting of contiguous objects, to be accurate. Thus, the combination of high temporal and spatial locality helps grouped object servers because it allows the server to use a general purpose object grouping algorithm that utilizes client provided object group size hints.

Experiment 22: Network Speed

The purpose of this experiment is to assess whether sending groups of objects is still beneficial as the network speed varies. Thus, the spatial, access and temporal locality values have been set to 10, 90 and 50 respectively (same as in Experiment 19). Network speed has been varied (10Mbps, 100 Mbps and 155 Mbps). The message transmission overheads associated with these three speeds has been presented in Section 8.3.1. As can be seen in Figure 8.34, ObjSrv and HybSrv outperform PageHard and PageSoft for the entire range of network speeds because the latter schemes transfer badly clustered pages from the server to the clients. However, as the network speed increases, sending badly clustered pages becomes more competitive due to higher bandwidth and lower transmission costs.

Experiment 23: Page Size

The purpose of this experiment is to check whether a change in page size affects the relative ordering in the performance of the different algorithms. The setup is similar to experiment 19, except the page size is increased to 16K. Figures 8.35 and 8.36 show the performance when the access locality is good and bad, respectively. Similar to Experiment 19, Figure 8.35 shows that when the access locality is good and the spatial locality is bad (10%), ObjSrv and HybSrv outperform PageSoft. Furthermore, similar to Experiment 20, when both access locality and spatial locality are bad, PageSoft and HybSrv

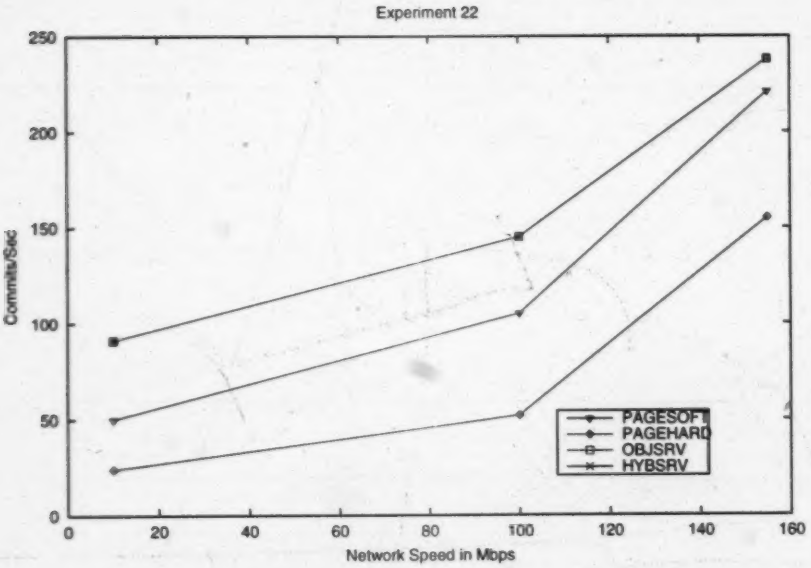


Figure 8.34: Varying Network Speeds

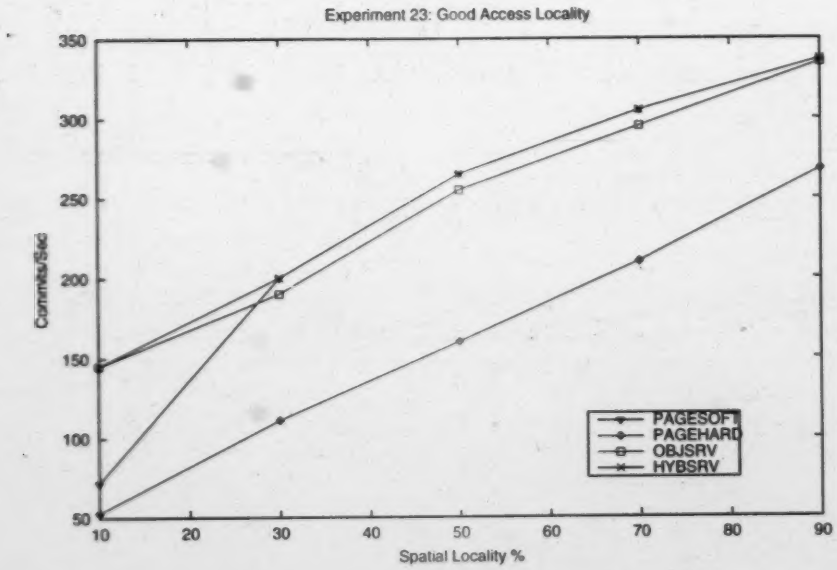


Figure 8.35: Large Page Good Access Locality

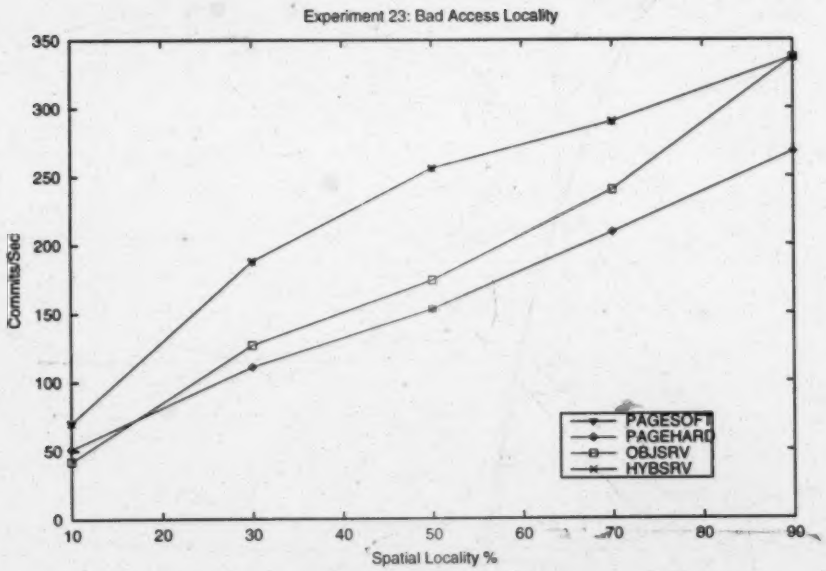


Figure 8.36: Large Page Bad Access Locality

outperform ObjSrv. Thus, a change in the page size has not resulted in a change in the relative ordering of the performance of the different algorithms.

8.2.4 Large Client and Small Server Buffers

In Large/Small configuration, the server buffer is small and cannot hold the working sets of the active clients (contended server buffer), but the client buffer is large enough to hold the client's working set. The Large/Small experiment evaluates the different client to server data transfer mechanisms, assessing whether it is efficient to return log records, updated pages and log records, or switch between these options. In this experiment the client buffer is set at 12.5% of the database size and the server buffer size is varied between 10 and 1% of the database size. These experiments have been run using the Private workload configuration because the focus of these experiments is to assess the performance of client to server data transfer and server buffer management mechanisms. The network speed has been set at 100 Mbps. Write probability is varied in these experiments. The spatial, temporal and access localities have

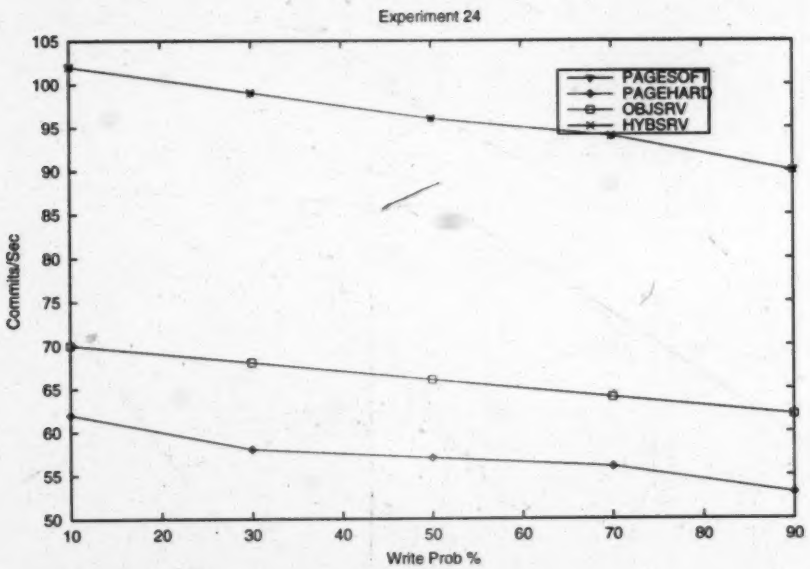


Figure 8.37: Server Buffer with Medium Contention

Experiment 24: 10 percent Write Probability

	Algorithms			
	PageHard	ObjSrv	PageSoft	HybSrv
Installation I/Os Per Commit	13.7	10.8	10.8	10.8

Figure 8.38: Installation I/Os

been set to 10%, 50% and 90%, respectively.

Experiment 24: Server Buffer with Medium Contention

The purpose of this experiment is to assess the impact of client to server data transfer mechanism on server buffer management. In this experiment, the server buffer size has been set to 10% of the database size. As shown in Figure 8.37, PageSoft, HybSrv, and ObjSrv outperform PageHard because they return updated objects to the server, whereas PageHard returns updated pages to the server. Storing updated objects in the MOB increases the absorption capability of the server buffer. That is, the MOB allows for the batched in-

stallation of the multiple updated objects to their corresponding disk pages. As shown in Figure 8.38, the MOB helps the architectures returning updated objects to have fewer installation I/O operations than PageHard. Objects belonging to the same page, that have been updated across multiple transactions by multiple clients can be installed together. However, in schemes that send updated pages to the server, if the data clustering is poor, then whole pages are stored at the server even if only a small portion of the page has been updated. This results in low server buffer utilization, and, thus, the writing of updated pages to disk interfere with the normal read disk traffic (to satisfy client read requests). Since HybSrv returns updated objects, its performance is also better than PageHard's performance.

Another key result of this experiment is that ObjSrv's performance trails PageSoft and HybSrv performance. Unlike experiment 19, PageSoft outperforms ObjSrv, because with a small server buffer, a client cache miss also results in a server cache miss. Thus, the importance of client cache management accuracy becomes more important when server buffer is small. HybSrv also outperforms ObjSrv because in HybSrv, the server realizes that its buffers are contended, and thus it sends pages to the clients to try to minimize the client cache misses.

Experiment 25: Highly Contended Server Buffer

The purpose of this experiment is to assess the impact of client to server data transfer mechanism on server buffer management when the server buffers are very contended. In this experiment, the server buffer size has been set to 1% of the database size. This experiment is supposed to represent the situation where the server buffer is extremely contended due to the simultaneous processing of multiple client requests. The server disk utilization varies between 90 and 95 percent. As shown in Figure 8.39, PageHard and HybSrv outperform ObjSrv and PageSoft because PageHard and HybSrv return updated pages to the server, whereas ObjSrv and PageSoft return updated objects to the server. In HybSrv, the server realizes that its buffers are contended and it sends a hint

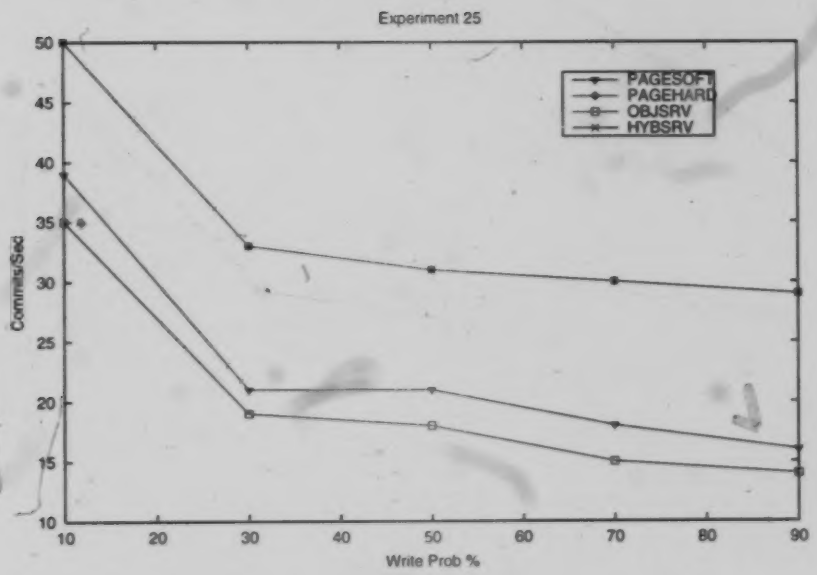


Figure 8.39: Highly Contended Server Buffer

Experiment 25: 10 percent Write Probability

Algorithms

	PageHard	ObjSrv	PageSoft	HybSrv
Installation I/Os Per Commit	14.2	18.4	18.4	14.2

Figure 8.40: Installation I/Os

to the client indicating that the client should return updated pages. Since the server buffer is contended, the MOB absorption capability is very low. As shown in Figure 8.40, in schemes that return updated objects to the server, there is a low chance for batching the installation of multiple object updates to a page. Therefore, the schemes returning updated objects perform a higher number of installation I/Os than the schemes returning updated pages to the server.

8.2.5 Small Client and Small Server Buffers

In Small/Small configuration, the server buffer is too small to hold the working sets of the active clients and the client buffer is small and it cannot hold the working set of the client. In this experiment, both the client and the server buffer sizes have been set to 1% of the database size. Since both server and client buffers are contended, the Small/Small buffer configuration helps to evaluate whether the client buffer or the server buffer has more impact on the overall performance. This experiment uses the Private data sharing pattern. The spatial locality, access locality and temporal locality values have been set to 10, 90 and 50%, respectively.

Experiment 26: Small-Small

The purpose of this experiment is to assess the relative importance of client and server buffers. As shown in Figure 8.41, PageSoft, HybSrv and ObjSrv outperform PageHard. Even though, as in Experiment 25, the server buffer configuration is very small, unlike Experiment 25, PageHard's performance trails the performance of the PageSoft and HybSrv because PageHard manages the client buffers strictly at page level. Since the spatial locality of the data access pattern is low, PageHard encounters low client buffer utilization. Thus, the gains made by PageHard due to the absence of installation reads at the server (during high server buffer contention) are mitigated due to the low client buffer utilization. Thus, this experiment shows that the efficiency of client buffer management is more important than the efficiency of the server buffer

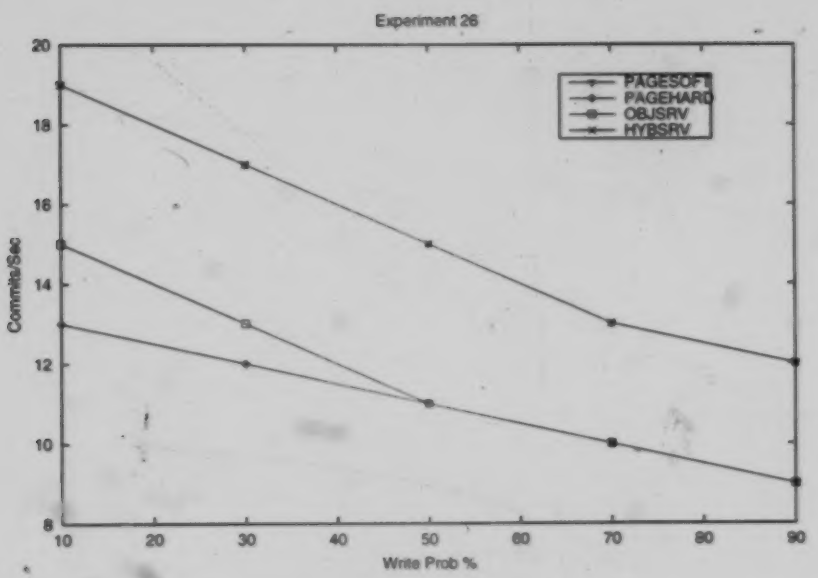


Figure 8.41: Small-Small

management. ObjSrv's performance trails PageSoft and HybSrv performance because of the high probability of a miss in the client cache being also a miss in the server cache. Since the temporal locality in this experiment is only 50%, ObjSrv incurs more client cache misses than PageSoft and HybSrv due to the inability of the server object grouping mechanism to construct accurate object groups.

In conclusion, this chapter presented a cache consistency performance study comparing AACC, ACBL and AOCC algorithms. This chapter also presented an integrated performance study comparing HybSrv, PageSoft, PageHard and ObjSrv architectures. The key findings of the cache consistency and the integrated performance studies are summarized in Chapter 9.

Chapter 9

Conclusions and Future Work

In this dissertation, a new *adaptive* server architecture has been proposed. This new server architecture incorporates new adaptive data transfer, cache consistency and recovery mechanisms. A prerequisite of adaptiveness is a hybrid server architecture that can efficiently handle both disk pages and logical objects. The hybrid server architecture incorporates a new concurrency control enhancement.

As shown by the performance study in Chapter 8, the existing client-server architectures and algorithms are not robust across different workloads and system configurations. Therefore, there is a need for adaptive algorithms which can dynamically adapt as the workload and system environment changes. Adaptive systems that minimize the system tuning and the configuration activities of programmers and system administrators have been identified as an important database system research area [Gra99, BBC+98, Ham99]. The work presented in this dissertation addresses this important research area within the context of client-server ODBMSs. The performance study presented in this dissertation verifies that it is possible to develop more robust adaptive algorithms and systems. The cache consistency study and the integrated performance study provide many interesting insights, some of which overturn commonly accepted beliefs. These are discussed below.

9.1 Cache Consistency Study Conclusions

A study that compares ACBL and AOCC cache consistency/concurrency control algorithms has shown that AOCC, which is an optimistic algorithm, outperforms ACBL, a pessimistic algorithm, even while encountering a high abort rate [AGLM95]. However, this dissertation has shown that even within the client-server ODBMS context, algorithms such as AACC can provide a low abort rate and can outperform high aborting algorithms, such as AOCC, with respect to overall system throughput.

AACC is an asynchronous cache consistency algorithm which outperforms the synchronous ACBL cache consistency algorithm. Previously, it was thought that synchronous cache consistency, such as Callback Locking (CBL) algorithms, outperform asynchronous cache consistency algorithms, such as No-Wait-Locking-Notify (NWL-Notify) [WR91], because asynchronous algorithms incur higher abort rates. In this dissertation it has been shown that an asynchronous algorithm such as AACC consistently outperforms a synchronous algorithm such as ACBL. The reason for this is that NWL-Notify is a detection-based algorithm and therefore encounters stale cache aborts. As an avoidance-based algorithm, AACC does not have this problem. Furthermore, NWL-Notify does not have an efficient abort processing mechanism as present in AOCC. It is the combination of optimistic detection-based and an inefficient abort processing mechanism allows CBL to outperform NWL-Notify.

AACC algorithm has a better combination of performance and abort rate than both ACBL or AOCC because it incorporates the following key enhancements:

- **Avoidance-Based:** AACC is an avoidance-based algorithm that, as stated above, does not encounter stale cache aborts. As shown in Chapter 8, the deadlock abort rate of AACC is much lower than the stale cache abort rate of AOCC. This, in turn, allows AACC to outperform AOCC for many key workloads and system configurations. Previously, detection-based asynchronous cache consistency algorithms were thought to be abort prone. However, AACC is avoidance-based and, therefore,

it does not encounter stale cache aborts. Furthermore, AACC uses new deadlock avoidance techniques which ensure that its deadlock abort rate is as low as ACBL's abort rate. This dissertation has shown that asynchronous messaging and avoidance-based notions are a good combination.

- **Shared/Private Regions:** The notions of shared and private page lock modes contribute to AACC's good performance when working with private workloads because they reduce the number of explicit messages. Unlike ACBL, which sends explicit lock escalation messages when updating pages that are only accessed by a single client, in AACC the server informs the clients that these pages are only present at the particular client (private lock mode), and thus the client piggybacks its lock escalation messages to the server.

- **Piggybacking Callback Messages:** In AACC, when a client receives a callback message, it sends an explicit callback response only if there is an object-level conflict. Otherwise, the client piggybacks its callback response to the server. Piggybacking of callback messages, in conjunction with piggybacking lock escalation messages for private pages, reduces the total number of messages sent between the client and the server. This helps AACC to outperform ACBL.

- **Asynchronous Messages:** AACC uses asynchronous lock escalation messages, which do not incur the blocking overhead common to systems that use synchronous lock escalation messages. The blocking overhead increases when the server and the network are heavily utilized. Asynchronous messages also reduce the number of deadlocks that are present in algorithms using deferred lock escalations. By sending the intent-to-update message to the server immediately, AACC reduces the window in which deadlocks can occur. One of the major drawbacks of deferred avoidance-based algorithms, such as O2PL [FC94], is that they incur a high deadlock rate due to deferring its lock escalation messages until

commit time.

- **Deadlock Avoidance Optimizations:** One of the drawbacks of moving away from using synchronous lock escalation messages to using asynchronous lock escalation messages is the increased possibility of deadlocks. AACC contains two deadlock avoidance optimizations which help it to maintain a deadlock abort rate that is similar to ACBL.

In addition to proposing the AACC algorithm for page servers, this dissertation has also adapted the AACC algorithm for object servers. Previously, data transfer and cache consistency/concurrency have been shown to be orthogonal to each other for page servers. In this dissertation, this orthogonality has been extended to object servers.

9.2 Integrated Study Conclusions

A new adaptive data transfer algorithm has been proposed in this dissertation, and its performance has been evaluated as part of the integrated performance study. The adaptive data transfer mechanism contains the following new features which help it to be robust, with respect to performance, as the workload and system configuration change:

- **Adaptive Server-to-Client Data Transfer:** This is the first dynamic data transfer mechanism to utilize an adaptive data transfer approach in both server-to-client, and client-to-server directions. The adaptive server-to-client data transfer mechanism helps to reduce the network overhead in situations where the clients access badly clustered pages. This optimization is very useful in low bandwidth environments, such as mobile networks and slow speed modem connections. The adaptive server-to-client data transfer mechanism can be used by page server architectures that employ a dual client buffer.

- **Adaptive Client-to-Server Data Transfer:** The adaptive client-to-server data transfer mechanism proposed here takes server buffer con-

tention level, client buffer management, and network cost into account when deciding whether to return updated pages or objects to the server. The previous client-to-server data transfer approaches did not take server buffer contention level into account [Ghe95, OS94a]. The adaptive client-to-server data transfer mechanism proposed here can be used by existing page server architecture systems.

- **Support for Varying Object and Page Sizes:** The previous object group forming mechanism [LAC+96] did not consider varying object sizes and page sizes into account, whereas the object group forming mechanism used by the adaptive data transfer mechanism handles varying object and page sizes. This object group forming mechanism can be used by existing grouped object server architectures.

- **Support for Varying Access Locality:** The previous object group forming mechanism [LAC+96] did not account for non-contiguous access to a page because the clients only kept track of the number of objects that have been accessed in the client cache, and did not care about the access locality characteristics. Therefore, the previous object group forming mechanism did not consider the notion of access locality; as a consequence, the performance of grouped object servers suffers during bad access locality. The adaptive data transfer mechanism presented here takes access locality characteristics into account and it uses this information to switch between requesting pages and object groups. This optimization can be used by the existing object server architectures.

A new object server recovery algorithm has also been proposed in this dissertation. All of the previous client-server recovery work has been conducted within the context of page servers. Moreover, this is also the first time that recovery issues have been studied for architectures where updates are performed both at the clients and the server.

The integrated performance study conducted in this dissertation has provided the following useful insights:

- It is desirable to have an adaptive data transfer architecture where pages and objects can be transferred both from the server to the client, and from the clients to the server.

- Concurrency control enhancements that allowed object servers to use AACC have ensured that object servers can efficiently use low aborting algorithms, and hence they can compete with page servers. Thus, concurrency control is not a liability for object servers. Previously, it was shown that object servers cannot efficiently use a pessimistic concurrency control algorithm [CFZ94].

- Previously, the redo-at-server recovery paradigm was shown to be unscalable [WD95]. In Chapter 8, it has been shown that a MOB allows the redo-at-server recovery paradigm to successfully compete with an ARIES-CSA style recovery mechanism which sends both log records and updated pages to the server.

- A previous study focusing solely on pointer swizzling [WD94] has shown that the hardware swizzling approach outperforms the software swizzling approach for most workloads. However, the integrated performance study presented in this dissertation has shown that the architectures using the software pointer swizzling approach outperform the architectures that employ hardware swizzling for most workloads and system configurations since the latter employ page-level client buffer management, page-level locking and page-level client to server data transfer mechanisms.

- Previously, it was shown that the object grouping mechanism allows grouped object servers to outperform page servers [LAC+96]. However, the performance study in Chapter 8 has shown that object grouping techniques that are executed at the server are only effective if the data access pattern has high access locality. Usually, the inefficiency of the server-based object grouping mechanisms leads to higher client cache miss rates, and this, in turn, leads to a greater number of object requests

being sent from the clients to the server. It is preferable to use dual page/object buffers at the client because they allow clients to have a high client cache hit rate even during periods of low access locality by caching pages, and they allow the clients to discard badly clustered pages when the page spatial locality is low.

- Previously, two separate studies on MOBs arrived at different conclusions with respect to whether it is beneficial to send updated pages or objects to the server. The initial study [OS94a] indicated that it is better to return updated pages to the server. The subsequent study [Ghe95] countered that it is better to return updated objects to the server. The results in this dissertation give the insight that the server buffer size is a key factor which determines whether it is desirable to return updated pages or objects to the server, thus, clarifying the previous results. If the server buffer size is very small then it is better to return updated pages, otherwise it is better to return updated objects.

9.3 Future Work

The research conducted in this dissertation can be extended in the following different ways:

- **Dynamic Dual Buffers:** The adaptive hybrid server architecture proposed in this dissertation does not employ dynamic dual buffers at the server and the client. The dynamic dual buffer at the client should automatically adjust the size of the page and object buffer partitions when encountering a workload change. Moreover, it should operate in conjunction with an adaptive data transfer mechanism. Similarly, a dynamic dual buffer at the server should combine the modified object buffer and the server page read buffer, and it should operate in conjunction with an adaptive data transfer mechanism. Furthermore, it is necessary to assess the impact on overall system performance due to these dynamic dual buffers.

- **Large Objects:** This dissertation has not dealt with large objects which span across multiple pages (e.g. multimedia or image objects). It is necessary to assess the impact of large object size on data transfer, cache consistency/concurrency control, recovery, buffer management and pointer swizzling algorithms. Extending the work in this dissertation for large objects would be a valuable contribution.

- **Mobile Environments:** Currently, the use of mobile devices is growing at a very rapid pace. Previously, mobile devices were primarily thought of as being capable of supporting the thin client architecture in which most of the processing is done at the server. However, with the continuous improvement in processing power, memory, and storage capacity of mobile devices, and with frequent disconnections of the mobile device from the network (and the need to maintain state information at the client) one can argue for the presence of thick mobile clients. In this scenario, one can modify and potentially use the client caching research that has been conducted in this dissertation. In the mobile domain, the algorithms presented herein have to be re-visited with respect to the broadcasting nature of the network medium, the lower bandwidth of the network medium, the absence of reliable connections, and the power supply constraints.

- **Hybrid Function-Shipping/Data-Shipping Systems:** This dissertation has concentrated on data-shipping systems algorithms and architectures. However, most commercial relational and object database management systems need to also support query processing. In many query processing cases it is beneficial to process the query at the server [KJF96]. Therefore, future database management systems need to provide support for hybrid function-shipping and data-shipping systems which support both navigational and set-oriented queries respectively. The data transfer, cache consistency, and recovery algorithms presented herein must be re-visited within the context of hybrid function-shipping/data-shipping systems.

- **Web Cache Consistency:** Currently, most of the cache consistency related research in the web domain is for configurations where the server generates new data, and it updates the client caches with the new updates. Most of the updates that are initiated by the clients are performed at the server similar to centralized database management systems. It would be interesting to examine whether the algorithms and techniques proposed in this dissertation can be modified and used in the web domain to improve the performance of update-oriented web applications by moving the work to the clients. Database applications have strict consistency requirements, which need to be relaxed for the web domain. Furthermore, the network model has to be changed to the WAN/Internet model in order to properly model the web domain. Finally, the simulator used in this dissertation has to be modified to handle hundreds (if not thousands) of clients, multiple servers, and proxy servers.

Bibliography

[ACL87] R. Agrawal, M. Carey, and M. Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4):609–654, December 1987.

[AFT97] L. Amsaleg, M. Franklin, and A. Tomasic. Dynamic Query Operator Scheduling for Wide-Area Remote Access. Technical Report CS-TR-381, University of Maryland, 1997.

[AGLM95] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of ACM SIGMOD Conference*, pages 23–34, 1995.

[BBC+98] P. Bernstein, M. Brodie, S. Ceri, M. Franklin, and et al. Asilomar Report. *ACM SIGMOD Record*, 27(4):74–80, December 1998.

[BDP92] F. Bancilhon, C. Delobel, and P.Kanellakis. *Building An Object-Oriented Database System.The Story of O2.* Morgan Kaufmann, 1992.

[BP95] A. Biliris and E. Panagos. A High Performance Configurable Storage Manager. In *Proceedings of ICDE*, pages 35–43, 1995.

[BP97] A. Biliris and E. Panagos. Synchronization and Recovery in a Client-Server Storage System. *VLDB Journal*, 6(3):209–223, August 1997.

[CALM87] M. Castro, A. Adya, B. Liskov, and Andrew Myers. HAC:Hybrid Adaptive Caching for Distributed Storage Systems. In *Proceedings*

of ACM Symposium on Operating System Principles, pages 102–115, 1987.

[CDF+94] M. Carey, D. DeWitt, M. Franklin, N. Hall, and et al. Shoring Up Persistent Applications. In *Proceedings of ACM SIGMOD Conference*, pages 383–394, 1994.

[CDKN94] M. Carey, D. DeWitt, C. Kant, and J. Naughton. A Status Report on the OO7 OODBMS Benchmarking Effort. In *Proceedings of OOPSLA Conference*, pages 414–426, 1994.

[CDN93] M. Carey, D. DeWitt, and J. Naughton. The OO7 Benchmark. In *Proceedings of ACM SIGMOD Conference*, pages 12–21, 1993.

[CFLS91] M. Carey, M. Franklin, M. Livny, and E. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architectures. In *Proceedings of ACM SIGMOD Conference*, pages 357–366, 1991.

[CFZ94] M. Carey, M. Franklin, and M. Zaharioudakis. Fine Grained Sharing in a Page Server OODBMS. In *Proceedings of ACM SIGMOD Conference*, pages 359–370, 1994.

[CK89] E. Chang and R. Katz. Exploiting Inheritance and Structure Semantics for Effective Clusteringand Buffering in an Object-Oriented DBMS. In *Proceedings of ACM SIGMOD Conference*, pages 348–357, 1989.

[CKV93] K. Curewitz, P. Krishnan, and J. Vitter. Practical prefetching via data compression. In *Proceedings of ACM SIGMOD Conference*, pages 257–266, 1993.

[CLH97] I. Chung, J. Lee, and C. Hwang. A Contention Based Dynamic Consistency Maintenance Scheme For Client Cache. In *Proceedings of CIKM*, pages 363–370, 1997.

[CS92] R. Cattell and J. Skeen. Object Operations Benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, April 1992.

[DFB+96] S. Dar, M. Franklin, B.T.Jonsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proceedings of VLDB Conference*, pages 330–341, 1996.

[DFMV90] D. DeWitt, P. Futtersack, D. Maier, and F. Velez. A study of three alternative workstation-server architectures for OODBS. In *Proceedings of VLDB Conference*, pages 107–121, 1990.

[FC94] M. Franklin and M. Carey. Client-Server Caching Revisited. *Distributed Object Management*, pages 57–78, 1994.

[FCL92] M. Franklin, M. Carey, and M. Livny. Global Memory Management in Client-Server DBMS. In *Proceedings of VLDB Conference*, pages 596–609, 1992.

[FCL97] M. Franklin, M. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Transactions on Database Systems*, 22(3):315–363, December 1997.

[Fra93] M. Franklin. *Caching and Memory Management in Client-Server Database Systems.* PhD thesis, University of Wisconsin-Madison, 1993.

[FZT+92] M. Franklin, M. Zwilling, C.K. Tan, M. Carey, and D. DeWitt. Crash Recovery in Client-Server EXODUS. In *Proceedings of ACM SIGMOD Conference*, pages 165–174, 1992.

[Gbu96] P. Gburzynski. *Protocol Design for Local and Metropolitan Area Networks.* Prentice-Hall, 1996.

[Ghe95] S. Ghemawat. *The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases.* PhD thesis, MIT, 1995.

[GK94] C. Gerlhof and A. Kemper. A Multi-Threaded Architecture for Prefetching in Object Bases. In *Proceedings of EDBT Conference*, pages 351–364, 1994.

[Gra99] J. Gray. What Next? A dozen remaining IT problems. In *Turing Award 1999 Lecture: www.research.microsoft.com/ gray/*, 1999.

[Gru97] R. Gruber. *Optimism versus Locking: A Study of Concurrency Control For Client-Server Object-Oriented Databases.* PhD thesis, MIT, 1997.

[Ham99] J. Hamilton. Networked Data Management Design Points. In *Proceedings of VLDB Conference*, pages 202–206, 1999.

[HKU99] L. Haas, D. Kossmann, and I. Ursu. Loading a Cache with Query Results. In *Proceedings of VLDB Conference*, pages 351–362, 1999.

[KGBW90] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(13):109–124, March 1990.

[KGM91] T. Keller, G. Graefe, and D. Maier. Efficient Assembly of Complex Objects. In *Proceedings of ACM SIGMOD Conference*, pages 148–157, 1991.

[KJF96] D. Kossmann, B.T. Jonsson, and M. Franklin. A Study of Query Execution Strategies for Client-Server Database Systems. In *Proceedings of ACM SIGMOD Conference*, pages 149–160, 1996.

[KK94] A. Kemper and D. Kossmann. Dual-Buffering Strategies in Object Bases. In *Proceedings of VLDB Conference*, pages 427–438, 1994.

[KPH98] K. Keeton, D. Patterson, and J. Hellerstein. A Case for Intelligent Disks (IDISKs). *ACM SIGMOD Record*, 27(3):42–52, August 1998.

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, and et al. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proceedings of ACM SIGMOD Conference*, pages 318–329, 1996.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[MN94] C. Mohan and I. Narang. ARIES/CSA: A Method for Database Recovery in Client-Server Architectures. In *Proceedings of ACM SIGMOD Conference*, pages 55–66, 1994.

[Obj98] Objectivity. White Paper: Choosing an Object Database. In *www.objectivity.com/ObjectDatabase/WP/Choosing/Choosing.html*, 1998.

[OS94a] J. O'Toole and L. Shrira. Hybrid caching for large scale object systems. In *Proceedings of Workshop on Persistent Object Systems*, pages 99–114, 1994.

[OS94b] J. O'Toole and L. Shrira. Opportunistic Log: Efficient Installation Reads in Reliable Object Server. In *Proceedings of the First Usenix Symposium on Operating Systems Design and Implementation OSDI*, pages 39–48, 1994.

[ÖV99] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition, 1999.

[ÖVU98] M.T. Özsu, K. Voruganti. and R. Unrau. An Asynchronous Avoidance-based Cache Consistency Algorithm for Client Caching DBMSs. In *Proceedings of VLDB Conference*, pages 440–451, 1998.

[PBJR96] E. Panagos, A. Biliris, H. Jagadish, and R. Rastogi. Fine-granularity Locking and Client-Based Logging for Distributed Architectures. In *Proceedings of EDBT Conference*, pages 388–402, 1996.

[PZ91] M. Palmer and S. Zdonik. Fido: A Cache That Learns to Fetch. In *Proceedings of VLDB Conference*, pages 255–264, 1991.

[SCO90] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. In *Proceedings of Winter Usenix Technical Conference*, pages 313–324, 1990.

[Sof98] Ardent Software. O2 engine. In *http://www.ardentsoftware.com/object/product/engine/index.html*, 1998.

[TN92] M. Tsangaris and J. Naughton. On the performance of object clustering techniques. In *Proceedings of ACM SIGMOD Conference*, pages 144–153, 1992.

[Ver98] Versant. Versant Delivers Next Generation Suite of Database Products. In *http://www.versant.com*, 1998.

[VÖU99] K. Voruganti, M.T. Özsu, and R. Unrau. An Adaptive Hybrid Server Architecture for Client Caching ODBMSs. In *Proceedings of VLDB Conference*, pages 150–161, 1999.

[WD94] S. White and D. DeWitt. QuickStore: A high performance mapped object store. In *Proceedings of ACM SIGMOD Conference*, pages 395–406, 1994.

[WD95] S. White and D. DeWitt. Implementing Crash Recovery in QuickStore: A Performance Study. In *Proceedings of ACM SIGMOD Conference*, pages 187–198, 1995.

[Whi94] S. White. *Pointer Swizzling Techniques for Object-Oriented Database Systems.* PhD thesis, University of Wisconsin-Madison, 1994.

[WN90] K. Wilkinson and M. Neimat. Maintaining Consistency of Client-Cached Data. In *Proceedings of VLDB Conference*, pages 122–133, 1990.

[WR91] Y. Wang and L. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. In *Proceedings of ACM SIGMOD Conference*, pages 367–376, 1991.

[ZC98] M. Zaharioudakis and M. Carey. Hierarchical, Adaptive Cache Consistency in a Page ServerOODBMS. *IEEE Transactions on Computers*, 47(4):427–444, 1998.

Appendix A

Glossary

This glossary contains a description of the acronyms that are used in this dissertation.

- **AACC:** refers to Asynchronous avoidance-based cache consistency algorithm that has been developed in this dissertation.
- **ACBL:** refers to Adaptive callback locking cache consistency algorithm developed at University of Wisconsin-Madison. It is a synchronous avoidance-based algorithm.
- **ACID:** refers to the transaction properties of atomicity, consistency, isolation and durability together are known as ACID properties.
- **AOCC:** refers to Adaptive optimistic cache consistency algorithm developed at MIT. It is a deferred detection-based algorithm.
- **ARIES:** refers to the recovery algorithm for centralized DBMSs developed at IBM Almaden. ARIES stands for Algorithm for Recovery and Isolation Exploiting Semantics.
- **ARIES-CSA:** refers to the client-server version of the centralized ARIES algorithm developed at IBM Almaden. CSA stands for Client-Server ARIES.

- **ARIES-ESM:** refers to the client-server version of the centralized ARIES algorithm developed at University of Wisconsin-Madison. ESM stands for Exodus Storage Manager.

- **CBL:** refers to synchronous avoidance-based page level only cache consistency algorithm developed at ObjectStore.

- **MHB:** refers to the modified hybrid buffer present at the server. This buffer stores both the updated objects and updated pages returned by the clients.

- **MOB:** refers to the object only version of MHB.

- **RPT:** refers to the resident page table data structure. It is present both in client memory and in server memory. The server RPT stores information about the pages present in server cache and the client RPT stores information about the pages present in the client cache.

- **RPD:** refers to the individual entries corresponding to pages in the RPT. These entries are known as RPDs or resident page descriptors.

- **ROT:** refers to the resident object table data structure. It is present both in client memory and in server memory. The server ROT stores information about the objects present in the server cache and the client ROT stores information about the objects present in the client cache.

- **ROD:** refers to the individual entries corresponding to objects in the ROT. These entries are known as RODs or resident object descriptors.